Collected Whizbangs

Volume 2

Collected Whizbangs

Volume 2

by
Roy A. Sykes, Jr.

STSC, Inc.
2115 E. Jefferson Street
Rockville, Maryland 20852
301-984-5000

10 9 8 7 6 5 4 3 2

Printed in the United States of America.

Note to Readers

Herewith we present another dozen "Whizbangs" taken from the columns of *APL***PLUS*† Service *News*, STSC's corporate newsletter. This collection includes "Whizbangs" from July 1978 through October 1980. STSC reprints the "Whizbangs" so that readers can have a handy compendium of *APL* techniques and tutorials for reference, careful study, or just leisurely perusal.

The "Whizbang" was born in December 1975 from the desire of STSC's *APL***PLUS* Service representatives to have a column describing *APL* programming techniques and tricks. Just what constitutes the difference between technique and trick was not made clear.

As several readers have pointed out, yesterday's tricks have become today's techniques. Therefore, the main purpose of the "Whizbang" is to educate readers not only in the obvious aspects of *APL*, but also in its nuances.

Thus the "Whizbang" addresses a wide audience. We hope that neophytes can use it to develop their skills and that old dogs can still learn new tricks.

Of course, I just write the "Whizbangs". The real job of producing them falls in the capable hands of Beth Matsumune and Laurie Russell. They, along with other STSC people who edit and review the drafts, make the "Whizbang" intelligible. The job of collecting, reviewing, and producing this manual was handled by Deborah Richardson, Allen Rose, and Pam McFarland. I thank all of these people, and I thank you, the readers.

Roy A. Sykes, Jr.
Los Angeles, California
July 1981

Editor's Note: Throughout this volume, bold APL is used to indicate the portion of an executable example that is entered by the user. Many of the examples use predefined objects. If the reader wishes to execute these examples, he should assign the objects first.

† *APL***PLUS* is a service mark and trademark of STSC, Inc., registered in the United States Patent and Trademark Office.

Contents

ρ*A*+*B* Problem 1

Does It Exist? *(July/August 1978)* 2

Where Is It? *(September/October 1978)* 5

The "All But" Case *(November/December 1978)* 9

Numeric Sorts *(January/February 1979)* 13

Other Sorts *(March/April 1979)* 16

CSORTR Algorithm 21

)ORIGIN 0 *(May/June 1979)* 23

Error Checking *(July/August 1979)* 27

A Clever Sort *(September/October 1979)* 30

Arrays of Arrays *(November/December 1979)* 33

Expansive Thinking *(January/February 1980)* 38

Rotation *(March/June 1980)* 42

Rotation Problem—Results 46

Creative Indexing *(July/October 1980)* 47

ρ*A*+*B* Solution 51

ρ*A*+*B* Problem

In October 1977 we presented a problem to readers. You might want to try your hand at it.

The shape of *A*+*B* is not all that easy to determine. Write a single statement that calculates the result of ρ*A*+*B* in as few characters as possible with the following constraints:

1. It works in either origin.
2. It uses no scalar dyadic functions*, nor any functions derived from them (that is, no reduction, scan, or inner or outer products).
3. It does not use execute (⍎), assignment (←), or diamond (⋄).
4. It uses no named objects other than *A* and *B*.

The record is the 43-character solution presented at the end of this volume. If you find a shorter one, let us know.

* By the way, can you name all 21 scalar dyadic functions?

Does It Exist?

A common problem in any computer language is determining if an object exists in a group of objects. For example, "Is 1403 one of our customer numbers?" or, "Does part number A4-103478-Y exist?" This article discusses some techniques for searching collections of objects (e.g., vectors and matrices) for possible member objects.

The most fundamental operation is an equality comparison between two scalars:

```
      1403=1374
0
```

If we extend either argument to a vector, we get a corresponding Boolean vector indicating whether or not the elements of the arguments match:

```
      1403= 1374 143 1403 1304 1403 2605
0 0 1 0 1 0
```

By performing an and-reduction (∧/) on such a Boolean vector, we answer the question "Is it always true that ...?" or "Is it always the case that ...?":

```
      ∧/1403= 1374 143 1403 1304 1403 2605
0
```

Conversely, or-reduction (∨/) answers the question "Is it ever true that ...?" or "Is there any case where ...?":

```
      ∨/1403= 1374 143 1403 1304 1403 2605
1
```

The or-reduction case is more directly coded by using the membership function (∈):

```
      1403∈ 1374 143 1403 1304 1403 2605
1
```

The above statement can be read as "Is 1403 an element (member) of the right argument?"

Membership also handles multiple elements on the left and returns a corresponding Boolean array:

```
      1403 1600 2605 ∈ 1374 143 1403 1304 1403 2605
1 0 1
```

Thus, the statement ∧/,*A*∈*B* answers the question "Is every element of *A* contained in *B*?" or "Is *A* a subset of *B*?"

We can extend the original case of ∨/*SCALAR*=*VECTOR* to arrays of arbitrary rank by using outer product equal (∘.=):

```
      ∨/A∘.=,B
```

The above statement is functionally identical to $A \in B$. Membership is typically more efficient, however, because it can stop searching for a given element once it has located the element in the right argument. Outer product must compare all elements against each other.

A list of customer numbers is usually stored as a vector wherein each element (a scalar) represents a single customer number; membership treats this case directly. Suppose, however, that the objects we're searching for are not scalars, but vectors. For example, a list of names is usually stored as a matrix wherein each row (a vector) represents a single name:

```
      ⍴NAMES ◇ NAMES
7 9
HAGENBUCH
CATELAIN
GRAZULIS
SMITH
MATSUMUNE
GARDNER
GRAZULIS
```

If we use membership to determine whether the name *JAKOBY* appears in the list, we'll find that it doesn't work since each element of the result is marked 1 if its corresponding element in the left argument is found *anywhere* in the right argument.

```
      'JAKOBY'∈NAMES
0 1 0 0 1 0
```

What we really want to do is compare the elements of 9↑'*JAKOBY*' with the corresponding elements of each row and perform an and-reduction to assure that all positions are matched:

```
      ⎕←L←¯1↑⍴NAMES
9

      ∧/NAMES=(⍴NAMES)⍴L↑'JAKOBY'
0 0 0 0 0 0 0
```

A more direct and efficient way to do this is to use an ∧.= inner product:

```
      NAMES∧.=L↑'JAKOBY'
0 0 0 0 0 0 0

      NAMES∧.=L↑'GRAZULIS'
0 0 1 0 0 0 1
```

Thus we get one element for each row in the left argument, and we can enter

```
      ∨/NAMES∧.=L↑'SMITH'
1
```

to determine whether any names match the right argument (which must have the same number of elements as the left argument has columns).

What if we want to search for several names?

```
      ⍴SEV ◇ SEV
3 9
GRAZULIS
SUWARA
MATSUMUNE

      NAMES∧.=SEV
LENGTH ERROR
      NAMES∧.=SEV
      ∧
```

We find that, although both arguments have nine columns, we've somehow goofed. For any inner product (in this case $A \wedge . = B$), the conformability rule is

```
      CC←(¯1↓⍴1/A),1↓⍴1/B
      1∈CC,=/CC
```

That is, the last coordinate of the left argument and the first coordinate of the right argument (the *combining coordinates*) must be equal, or either may be 1, or either argument may be a scalar. (Note that 1/ ravels scalars to one-element vectors, but leaves all other arrays unchanged.) The shape of the result is (¯1↓⍴A), 1↓⍴B. That is, after the two combining coordinates are checked for conformability, they are processed and deleted.

Let's try getting the two coordinates of equal length together with transpose (⍉):

```
      ⍴⍉SEV
9 3

      NAMES∧.=⍉SEV
0 0 0
0 0 0
1 0 0
0 0 0
0 0 1
0 0 0
1 0 0
```

It appears that we've compared all seven rows of *NAMES* against all three rows of *SEV*, much like

```
      A∘.=B
```

compares all elements of *A* against all elements of *B*. Performing an or-reduction along the first coordinate returns the desired result:

```
      ∨⌿NAMES∧.=⍉SEV
1 0 1
```

Alternatively, we can interchange arguments and perform an or-reduction along the last dimension:

```
      SEV∧.=⍉NAMES
0 0 1 0 0 0 1
0 0 0 0 0 0 0
0 0 0 0 1 0 0

      ∨/SEV∧.=⍉NAMES
1 0 1
```

To generalize somewhat, given a vector *VEC*, the expression

```
∨/A∘.=VEC
```

Equivalent to *A∈VEC*

determines whether the individual elements (scalars) of array *A* exist in *VEC*. The result has shape ⍴*A*. The parallel expression for matrices is

```
∨/A∧.=⍉MAT
```

which, given a matrix *MAT*, determines whether the individual rows (vectors) of array *A* (having the same number of columns as *MAT*) exist in *MAT*. The result has shape ¯1↓⍴*A*.

The "Where Is It?" article which follows discusses techniques for determining not only *whether* an object exists in an array, but *where* it exists. See if you can anticipate me by writing a matrix equivalent for dyadic iota (*VEC*⍳*A*).

Where Is It?

In the previous article, "Does It Exist?", we discussed determining if an object (for instance, a scalar or vector) exists in a group of objects (for instance, a vector or matrix). In this article we'll talk about techniques for locating an object in a group of objects. With these techniques, we'll be able to determine not only if an object exists, but also where it is located. We'll use origin 1 (⎕IO←1) throughout.

There are two common interpretations for locating an object:

- Where is the first occurrence?
- Where are all occurrences?

Given all occurrences, we can always select the first. However, if we want only the first occurrence, we can often use a more direct technique.

In the following example, the equal function (=) gives us a Boolean representation of all places where 1403 occurs in the vector. We convert this to indices using compression on all indices of the vector (⍳⍴VEC), and then select the first index:

```
      VEC← 1374 143 1403 1304 1403 2605
      SCAL←1403
      ⎕←B←SCAL=VEC
0 0 1 0 1 0

      ⎕←I←B/⍳⍴VEC
3 5

      1↑I
3
```

Thus 1↑(SCAL=VEC)/⍳⍴VEC returns the index of the first match (or 0 if there are no matches).

The position of the first match can also be represented as a Boolean vector by using less-than scan (< \). Less-than scan leaves the first 1 in the Boolean vector as 1 and turns all other 1's to 0's; the 0's in the vector are unaffected.

```
      B
0 0 1 0 1 0

      <\B
0 0 1 0 0 0

      (<\B)/⍳⍴VEC
3
```

Hence we can also compute the index of the first match by $(<\backslash SCAL=VEC)/\iota\rho VEC$ (this time returning an empty vector if there are no matches).

A more innovative approach, suggested to me by Jordan Plitteris of Clairol, is to count the number of leading *mis*matches (0's in the Boolean vector) and simply add 1 (or more generally, $\square IO$):

```
      ~SCAL=VEC
1 1 0 1 0 1

      ∧\~SCAL=VEC
1 1 0 0 0 0

      1++/∧\~SCAL=VEC
3
```

Using the identity

$$\sim A=B \leftrightarrow A\neq B$$

we can restate it as:

```
      1++/∧\SCAL≠VEC
3
```

This algorithm returns one plus the number of elements in the vector $(1+\rho VEC)$ if $SCAL$ is not found anywhere in VEC.

We must admit we've been teasing you a little (for good reason as we shall see), because all of our solutions simply reduce to a case of dyadic iota (ι):

```
      VEC⍳SCAL
3
```

For each element of its right argument, dyadic iota returns the index of its first match in the vector left argument. If an element from the right argument is not located in the left argument, the corresponding element of the result is $1+\rho VEC$.

```
      ⍴VEC
6

      VEC⍳ 1403 2605 1999 1403 143
3 6 7 3 2
```

Thus, dyadic iota directly handles the problem of locating the first occurrence of many elements in a vector. You may notice that $(VEC=SCAL)\iota 1$ is equivalent to $VEC\iota SCAL$.

We can generalize the $<\backslash$ and $\neq$ expressions to emulate dyadic iota by accepting many elements and returning one plus the last index if not found. The fundamental change is substituting outer products $\circ.=$ and $\circ.\neq$ for the scalar functions = and $\neq$. In the following code, A represents any array and VEC is the vector in which we are searching. The result has shape ρA.

```
NF←1+⍴VEC ⋄ SA←⍴A              Value if not found; shape of A.
VEC⍳A ↔ SA⍴(,<\(A∘.=VEC),1)/(×/SA,NF)⍴⍳NF
      ↔ 1++/∧\A∘.≠VEC
```

Observe that we must catenate a 1 to the result of the $A\circ.=VEC$ outer product to assure that the less-than scan always marks an element in each row. The leading mismatch algorithm is certainly more elegant, and executes faster and more compactly.

What if we want to find the first occurrence of a vector in a matrix? If we have a list of names stored as a matrix,

```
      ⍴NAMES ◇ NAMES
7 9
HAGENBUCH
CATELAIN
GRAZULIS
SMITH
MATSUMUNE
GARDNER
GRAZULIS
```

and we attempt to find the row index of a single name, we are reminded that dyadic iota is restricted to vector left arguments.

```
      NAMES⍳'GRAZULIS'
RANK ERROR
      NAMES⍳'GRAZULIS'
      ^
```

Instead, we must use an ∧ . = inner product to compare each row with the right argument, and then, on this result, use dyadic iota to determine the first index or / ⍳ to determine all indices. Note that the length of the name must be equal to the length of each row (the number of columns in the matrix).

```
      ⎕←L←¯1↑⍴NAMES
9
      ⎕←BV←NAMES∧.=L↑'GRAZULIS'
0 0 1 0 0 0 1
      BV⍳1 ◇ BV/⍳⍴BV
3
3 7
```

A further complication arises if we wish to find the first occurrence of several names:

```
      ⍴SEV ◇ SEV
3 9
GRAZULIS
SUWARA
MATSUMUNE
      ⎕←BM←SEV∧.=⍉NAMES
 0 0 1 0 0 0 1
 0 0 0 0 0 0 0
 0 0 0 0 1 0 0
      BM⍳1
RANK ERROR
      BM⍳1
      ^
      BM/⍳⍴BM
RANK ERROR
      BM/⍳⍴BM
      ^
```

BM is a matrix, and therefore can't be used as the left argument to iota or compression.

Instead, we can apply the same techniques used above to emulate dyadic iota for vectors, but this time we must use inner products ∧ . = and ∨ . ≠ rather than outer products ∘ . = and ∘ . ≠.

In the following code, *A* represents any array having the same number of columns as *MAT*, the matrix in which we are searching. The result has shape ¯1↓⍴*A*, the logical shape of the data we're locating.

```
      NF←1+1↑⍴MAT ◇ SA←¯1↓⍴A
      SA⍴(,<\(A∧.=⍉MAT),1)/(×/SA,NF)⍴⍳NF
```

or better yet

```
      1++/∧\A∨.≠⍉MAT
```

In conclusion, we have seen that by deriving identities for an existing *APL* primitive, we can generalize these identities to accommodate cases not directly handled by the primitive.

Using □*SS*

In "Character Searches" (*Collected Whizbangs,* Volume 1), we solicited contributions from readers on interesting uses of the system function □*SS*. □*SS* locates all matches of its right argument in its left argument, returning a Boolean vector.

```
      'MISSING MISSISSIPPI' □SS 'ISSI'
0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
```

As we've seen, we can apply < \ to the result to flag only the first match.

```
      <\'MISSING MISSISSIPPI' □SS 'ISSI'
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Or, we can reverse twice to flag only the last match.

```
      ⌽<\⌽'MISSING MISSISSIPPI' □SS 'ISSI'
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
```

Marty Zimelis (STSC) points out that □*SS* can be used to delete multiple adjacent spaces from a character vector:

```
      C←'  SO BE IT IF   WE DO  AS IN TV  '
      (~C □SS '  ')/C
SO BE IT IF WE DO AS IN TV
```

The "All But" Case

Note: `⎕IO←1` *throughout.*

Most people think positively. For instance, most would ask "How many customers pay on time?" rather than "How many customers are not delinquent?" Similarly, an *APL* programmer is more likely to write `(BAL>0)/CUSTNOS` than `(~BAL≤0)/CUSTNOS` to select all customers with positive balances. But occasionally programmers are asked to solve "all but" problems, forcing them to think negatively.

Fortunately, *APL* provides facilities to allow either direct coding of negative inquiries, or their transformation to positive requests. The example above is illustrative. Perhaps the most direct "all but" function is drop (↓). The statement `5 ¯2 ↓MATRIX` can be interpreted as "Return all but the first 5 rows and last 2 columns of *MATRIX*".

One characteristic of most practical "all but" problems is a limited universe of discourse. In the two examples above, *CUSTNOS* and *MATRIX* represented our universe of discourse. On the other hand, although we can generate the first five positive integers (`⍳5`), we can't generate all but the first five positive integers (`5↓⍳⌊/⍳0`) because, practically speaking, it's an infinite set.

We normally need a fixed and manipulable population which can be divided into two groups—the "haves" and the "have-nots", if you will.

Some Examples

Consider the following: Sales regions are stored in a database as arbitrary single-character codes. There are two variables used to interpret the codes (`REGNAMES[REGCODES⍳CODES;]`):

- *REGCODES*, a vector of the distinct codes;
- *REGNAMES*, an associated matrix of region names.

The problem is that Marketing reorganizes and adds three new regions, and we must assign the next three sales region codes.

The practical solution is

```
NEWREGCODES←3↑(~⎕AV∈REGCODES)/⎕AV
```

because `⎕AV` is small enough to manipulate and is divisible into two groups. The interpretation is "Return the first 3 elements of all characters except those already used". But what if *REGCODES* contained positive integers

instead of characters? Even if we could generate all positive integers (*API*←⍳⌊/⍳0), it would be monstrously impractical to attempt

```
NEWREGCODES←3↑(~API∈REGCODES)/API
```

Instead we would resort to some other algorithm, such as

```
NEWREGCODES←(⍳3)+⌈/0,REGCODES
```

A similar problem is finding an arbitrary tie number for a file. The tie number will be one of the positive integers that is not already tied to a file, as given by ⎕*FNUMS*. Our universe of discourse is all integers from 1 to ¯1+2*31, which is plainly too big to manipulate. One approach is

```
TIENO←1+⌈/0,⎕FNUMS
```

but this fails if (¯1+2*31)∈⎕*FNUMS*—that is, if another file has already been tied to the largest allowable value.

The solution is to limit the universe of discourse to some practical size. Consider: if there are *N* files tied, at least one of the numbers ⍳*N*+1 is not being used as a tie number. Thus we can use ⍳1+⍴⎕*FNUMS* as our universe:

```
U←⍳1+⍴⎕FNUMS ◇ TIENO←1↑(~U∈⎕FNUMS)/U
```

Or, we can be more succinct by using dyadic iota (⍳) to automatically return the next higher integer if required:

```
TIENO←((⍳⍴⎕FNUMS)∈⎕FNUMS)⍳0
```

"All But" Selection

For compression (/) and take and drop (↑ and ↓), "all but" selection is easy:

B/*VEC*	All elements where *B*=1.
(~*B*)/*VEC*	All elements except those where *B*=1 (i.e., where *B*=0).
N↑*VEC*	The first (or last, if *N*<0) *N* elements.
N↓*VEC*	All but the first (or last) *N* elements.

However, subscription doesn't have a direct "all but" formulation. Indices represent not only selection criteria but also ordering, replication, and structure information. For example, what is the "all but" case in the following example?

```
      VEC←'BUSTLE'
      INDS← 2 4 ⍴ 4 6 5 5 1 6 5 4
      VEC[INDS]
TELL
BELT
```

Plainly we have to form some rules for "all but" indexing.

Let us define "all but" indexing as selecting elements from an array in the order in which they occur in the array. Further, let us state that each element selected is exactly one of those whose indices do not occur in the given set of indices. It all adds up to a case of compression:

```
      VEC[(~(⍳⍴VEC)∈INDS)/⍳⍴VEC]      Select all but INDS.
US
```

Or, more directly,

```
      (~(⍳⍴VEC)∈INDS)/VEC
US
```

Although the latter case is preferable for selection, the former is necessary for assignment:

```
      VEC[(~(⍳⍴VEC)∈INDS)/⍳⍴VEC]←'E'
      VEC
BEETLE
```

If ρ*VEC* is large, the epsilon (∈) search for ⍳ρ*VEC* can be expensive. Thus an alternate approach for computing the compression vector is desirable:

```
      ALLBUT←(ρVEC)ρ1 ◇ ALLBUT[INDS]←0
```

We construct a Boolean vector that selects all elements of *VEC*. We then assign 0's to the indices we do not want to select (the same element of *ALLBUT* may be assigned several 0's). *ALLBUT* is identical to the original compression vector.

```
      ALLBUT ◇ ~(⍳ρVEC)∈INDS
0 1 1 0 0 0
0 1 1 0 0 0

      ALLBUT/VEC
EE

      VEC[ALLBUT/⍳ρVEC]←'AT'
      VEC
BATTLE
```

Another Application

The following technique pertains to □*FSTAC* (setting file access), but may also apply to □*IDLIST* or any other function employing similarly-coded arguments.

The permission codes in column 2 of □*FSTAC* are based on the sum of access codes, which are powers of 2. There is a fixed number of distinct access codes at any one time (currently *N*←14), but *N* may increase in the future if new types of file access are defined. There are two approaches to setting file access:

(1) Grant only the given access (additive):

```
      +/CODES
```

This technique is simple. Just add the desired access codes to compute the permission code. For example, to give □*FREAD* (1), □*FAPPEND* (8), and □*FRESIZE* (1024) access, use the code 1033.

(2) Grant all but the given access (subtractive):

```
      ¯1-+/CODES
```

This approach requires that you either know *N* (the number of distinct access codes that make up the universe of discourse) and add all codes other than those you wish to grant, or that you know that permission codes are actually interpreted as *PC*←(2*N)|*PC*.

To use the additive technique to give all but □*FERASE* (4) and □*FDROP* (32) access, you have to add up the codes for everything else!

```
      +/ 1 2  8 16  64 128 256 512 1024 2048 4096 8192
16347
```

The subtractive technique is more direct, and you don't have to know *N*:

```
      ¯1-+/ 4 32
¯37

      (2*N)|¯37
16347
```

We see that a permission code of ¯37 is equivalent to 16347. The subtractive technique has the advantage of being independent of the number of types of access available. Moreover, it automatically grants any future access (as when *N* increases to enable more types of access). I often give myself ¯5 access (all

but □*FERASE*) to my own files to force me to perform an additional safety step before erasing a file (that of resetting the access matrix).

An Unusual Application

Finally, an unusual but useful "all but" application is determining all functions in a workspace except those specified. For example, if we have a group defined that contains the names of all subroutines required by a particular function, before invoking that function we may want to erase all other functions to increase working storage (□*WA*). Our universe of discourse is the names of all functions in the workspace; we want to return all except those contained in the group specified in the argument.

The traditional approach is clumsy:

```
     ∇ TOGO←ALLFNSBUTGRP TOKEEP;COLS;ALLFNS
[1]    ⍝ RETURNS A MATRIX OF THE NAMES OF ALL FUNCTIONS
[2]    ⍝ IN THE WORKSPACE EXCEPT THOSE IN GROUP <TOKEEP>.
[3]    ALLFNS←□NL 3 ◇ TOKEEP←□GRP TOKEEP
[4]    COLS←(1↓⍴ALLFNS)⌈1↓⍴TOKEEP
[5]    ALLFNS←((1↑⍴ALLFNS),COLS)↑ALLFNS
[6]    TOKEEP←((1↑⍴TOKEEP),COLS)↑TOKEEP
[7]    TOGO←(∧/ALLFNS∨.≠⍉TOKEEP)⌿ALLFNS
     ∇
```

On line [3] we obtain the two lists of functions. Lines [4] to [6] are devoted to housekeeping—making sure the matrices are conformable for the ∨.≠ inner product on line [7]. Line [7] searches and returns all function names in the workspace that do not occur in the list of names to keep.

The innovative approach is truly bizarre:

```
     ∇ TOGO←AFBG TOKEEP;LOCALFN
[1]    ⍝ RETURNS A MATRIX OF THE NAMES OF ALL FUNCTIONS
[2]    ⍝ IN THE WORKSPACE EXCEPT THOSE IN GROUP <TOKEEP>.
[3]    TOKEEP←,';',□GRP TOKEEP
[4]    TOKEEP←'LOCALFN;LOCALFN',TOKEEP
[5]    ⍎□FX TOKEEP,[0.5](⍴TOKEEP)↑'TOGO←□NL 3'
     ∇
```

AFBG defines a local function that localizes the names of all objects to keep. When the local function is executed, it returns the names of all functions defined at that moment, which excludes the localized names in the header! Note that, because of shadowing, neither *ALLFNSBUTGRP* nor *AFBG* can return names of functions homonymous with names in their headers.

Numeric Sorts

While designing (in collaboration with Martin Gardner, STSC) and implementing the functions in workspace 6 *SORT* on the *APL*PLUS* System, we investigated the properties of the *APL* grading functions (⍋ and ⍒) and their application to sorting. This article presents a short tutorial on sorting (reordering) numeric arrays in *APL*. All examples are in origin 1 (⎕*IO*←1).

Sorting Vectors

Grade up (⍋) and down (⍒) are marvelous concepts, although their definition is currently limited to vectors. Rather than actually sorting a vector, they return the indices that would sort the vector. For example,

```
      VEC← 70 46 18 46 59 32 46 59
      ⎕←INDS←⍋VEC
3 6 2 4 7 5 8 1
```

The result is a permutation of ⍳⍴*VEC* indicating that the third element of *VEC* is the smallest, the sixth element is the next smallest, and so on. Consequently, we can use the resultant indices with subscription to rearrange (sort) the vector in ascending order:

```
      VEC[INDS]
18 32 46 46 46 59 59 70
```

Why doesn't grade order a vector directly, rather than requiring the subsequent indexing operation? The answer is that grade, being a more fundamental operation to express ordering, has a variety of other uses such as ranking and searching. You can also use its result to rearrange other arrays on the basis of the ordering of a corresponding vector. For example, given a matrix of data, you can order the rows of that matrix based on one column (in this example, the year's column):

```
      DATES
1978   10   31
1978   12    2
1979    1    5
1978   11   30
1979    1   16
1978   12   15
1978   11   30
1978   12    5
```

```
      DATES[⍋DATES[;1];]
1978   10   31
1978   12    2
1978   11   30
1978   12   15
1978   11   30
1978   12    5
1979    1    5
1979    1   16
```

Sorting Matrices on Keys

More often we want to sort a matrix based on several columns. In the previous example, we sorted based on only the first column (years) and ignored the second and third columns (months and days). To order *DATES* properly, we want to rearrange by year, and if year is identical, rearrange by month, and if month is identical, rearrange by day. Another way of saying this is that year is the major sort key, month is the intermediate key, and day is the minor key.

There are several ways to perform key sorts. One way is to sort by the minor key first, then by the intermediate keys, and finally by the major key:

```
     ∇ R←DATASORT MAT;COL;⎕IO
[1]   ⍝ NUMERIC MATRIX SORT.
[2]     ⎕IO←1 ◇ COL←(⍴MAT)[2]
[3]   A:→COL↓B ◇ MAT←MAT[⍋MAT[;COL];]
[4]     COL←COL-1 ◇ →A
[5]   B:R←MAT
     ∇
```

The above technique represents a brute-force approach, however. At each iteration we must rearrange the entire matrix; this requires a new copy of the matrix and consumes unnecessary computer time. The more sophisticated approach given below rearranges only the column being sorted and the resultant indices. It defers rearranging the entire matrix until the final indices have been computed:

```
     ∇ R←INDSORT MAT;COL;⎕IO
[1]   ⍝ SUBTLER NUMERIC MATRIX SORT.
[2]     ⎕IO←1 ◇ COL←(⍴MAT)[2] ◇ R←⍳1↑⍴MAT
[3]   A:→COL↓B ◇ R←R[⍋MAT[R;COL]]
[4]     COL←COL-1 ◇ →A
[5]   B:R←MAT[R;]
     ∇
```

```
      R←DATASORT DATES ◇ S←INDSORT DATES
      ∧/,R=S ◇ S
1
 1978   10   31
 1978   11   30
 1978   11   30
 1978   12    2
 1978   12    5
 1978   12   15
 1979    1    5
 1979    1   16
```

If *N* is the number of columns in the matrix, *DATASORT* subscripts *N*×*N*+1 columns of data, while *INDSORT* subscripts only *N*×3 columns of data. For a three-column matrix such as *DATES*, *INDSORT* subscripts 75 percent of the elements that *DATASORT* does; for a ten-column matrix, only 27 percent. *INDSORT* can be further refined by computing the grade indices of the minor key directly, rather than using the index vector (⍳1↑⍴*MAT*) for the first iteration. The function *NSORT* in workspace 6 *SORT* uses this technique.

Encoding Keys

An alternative approach to sorting numeric matrices that is typically faster, but less general, is to arithmetically combine the keys into a single vector. Because of finite numeric precision in the computer, this approach is usable only if the resultant keys are all less than 2*56 (approximately 7.2*E*16). A typical formulation for positive integers is

```
MAT[⍋(⌈/⍉MAT)⊥⍉MAT;]
```

This expression computes the minimum values that, when used as the left argument to base value (⊥), will uniquely represent rows of the matrix.

Most of the time, we know in advance the maximum values of each column, and we can embed constants directly into the code. With our *DATES* matrix, we might use either of the following as the argument to grade:

```
      100⊥⍒DATES
19781031 19781202 19790105 19781130 19790116 19781215 19781130 19781205

      0 12 31 ⊥⍒DATES
736157 736190 736224 736187 736235 736203 736187 736193
```

We map the key values into distinct numbers such that the minor key is represented in the low-order digits, the major key in the high-order digits, and the intermediate keys in between.

Controlling the Order

In all cases, the order of the sort can be controlled by using either ⍋ (for ascending order) or ⍒ (for descending order). Alternatively, because of the following identities, we can use only ⍋ by multiplying the data by 1 (for ascending order) or by ¯1 (for descending order).

```
⍒A  ←→  ⍋-A  ←→  ⍋A×¯1
```

In fact, we can sort columns in different orders using these identities. For example, to sort *DATES* in descending order by year, but in ascending order by month and day, we can use any of

```
      R←|INDSORT DATES×(⍴DATES)⍴¯1 1 1
      S←DATES[⍋ 0 ¯12 31 ⊥⍒DATES;]
      T←DATES[⍋DATES+.× ¯10000 100 1;]
      (∧/,R=S)∧∧/,R=T ◇ R
1
 1979     1     5
 1979     1    16
 1978    10    31
 1978    11    30
 1978    11    30
 1978    12     2
 1978    12     5
 1978    12    15
```

An Unusual Technique

Yet another technique for sorting an integer matrix *M* is the elegant algorithm shown below:

```
R←M,0 ◇ R←M[⍋+⌿R<.-⍉R;]
```

Terse, and completely general, it is nonetheless quite slow in most cases. (An exception is "short, fat matrices"—those having very few rows and many columns.)

The unusual inner product determines whether each row of *M* is lexicographically smaller (1) or the same or larger (0) than every other row. It does this by performing a less-than reduction (< /) on the arithmetic differences between all pairs of rows. The < / ignores all leading zeros except for the last element, thus requiring the catenation of 0 (actually, any numeric scalar will work) to the matrix being analyzed. If the first nonzero element is negative, the < / returns 1; otherwise it returns 0.

Other Sorts

In the previous article, "Numeric Sorts", we discussed sorting numeric matrices using the columns as keys. In this article we'll discuss sorting character data and sorting rows independently. As usual, all examples are in origin 1 (⎕IO←1) unless otherwise noted.

Sorting Character Vectors

Grade up (⍋) and down (⍒) currently operate only on numeric vectors. To sort character data, the data must first be transformed to numeric data. The normal technique is to use index-of (dyadic iota, ⍳) to map each character to its position within a defined collating sequence:

```
      LET←'FEEDBACK'
      LET[⍋LET]
DOMAIN ERROR
      LET[⍋LET]
           ^

      CS←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      CS⍳LET
6 5 5 4 2 1 3 11

      LET[⍋CS⍳LET]
ABCDEEFK
```

Sometimes the collating sequence is unusual: `'PMCE'` for continental United States time zones, `'CDEFGAB'` for the musical scale, `'PBNRQK'` for chess pieces, `'IVXLCDM'` for Roman numerals, and `'CDHS'` and `'23456789TJQKA'` for card suits and denominations. For example, let the following matrix represent a bridge hand:

```
      HAND
K45QA3Q8J2T7J
HSCCHDSSSHCSC
```

We can sort the hand by suit alone,

```
      HAND[;⍒'CDHS'⍳HAND[2;]]
4Q8J7KA235QTJ
SSSSSHHHDCCCC
```

by denomination alone,

```
      HAND[;⍒'23456789TJQKA'⍳HAND[1;]]
AKQQJJT875432
HHCSSCCSSCSDH
```

or by denomination within suit,

```
      D←⍋'23456789TJQKA'⍳HAND[1;]
      HAND[;D[⍋'CDHS'⍳HAND[2;D]]]
QJ874AK23QJT5
SSSSSHHHDCCCC
```

The last example illustrates the basic technique of sorting a character matrix by keys. The minor key is denomination (the first row) and the major key is suit (the second row). In this case, each key has its own distinct collating sequence, and we are rearranging columns.

Sorting Character Matrices

Most often we simply want to sort a matrix of words or names; that is, we want to rearrange rows. The collating sequence is usually the alphabet, often prefixed with a space and punctuation marks and suffixed with the numeric digits:

```
      CS
 -/ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

We can modify the *INDSORT* function presented in the previous article to accommodate character data by simply translating the characters to numbers on line [3]:

```
     ∇ R←CS CINDSORT MAT;COL;⎕IO
[1]    ⍝ CHARACTER MATRIX SORT.
[2]    ⎕IO←1 ◇ COL←(⍴MAT)[2] ◇ R←⍳1↑⍴MAT
[3]   A:→COL↓B ◇ R←R[⍋CS⍳MAT[R;COL]]
[4]    COL←COL-1 ◇ →A
[5]   B:R←MAT[R;]
     ∇

      WEST
HAWAII
ALASKA
WASHINGTON
OREGON
CALIFORNIA
NEVADA

      CS CINDSORT WEST
ALASKA
CALIFORNIA
HAWAII
NEVADA
OREGON
WASHINGTON
```

However, this approach tends to execute slowly because it must iterate for every column of the array. Since we know in advance that the largest number we may encounter is ⍴CS (assuming the collating sequence contains all possible characters), we can profitably encode several columns into one key using base value (⍳). By doing so we can reduce the number of iterations required. In fact, were it not for finite numeric precision within the computer, the following expression would sort the rows of any character matrix.

```
CHRMAT[⍋(⍴CS)⊥CS⍳⍉CHRMAT;]
```

But the largest unique integer that can be distinctly represented on the *APL*PLUS* System is 2*56. If the result of base value exceeds that, precision is lost and words like "individualist" and "individualism" become indistinguishable.

We must therefore calculate the maximum number of columns we can encode into a single integer. The derivation of *MNOC*, which must be integral, is as follows:

```
((ρCS)⊥MNOCρ¯1+ρCS) ≤ 2*56
    (¯1+(ρCS)*MNOC) ≤ 2*56
       ((ρCS)*MNOC) ≤ 1+2*56
       ((ρCS)*MNOC) ≤ 2*56
              MNOC ≤ (ρCS)⍴2*56
              MNOC = ⌊(ρCS)⍴2*56
```

```
      ρCS ◇ ⎕←MNOC←⌊(ρCS)⍴2*56
39
10
```

This formula assumes values from dyadic iota in the range 0 to ¯1+ρCS. These can be achieved by either performing the iota in origin 0, or subtracting 1 from the result of origin-1 dyadic iota.

Given that we can uniquely represent several columns of character data in one integer, we can modify the sorting algorithm to the following:

```
     ∇ R←CS CMSORT MAT;BASE;COL;COLS;MNOC;⎕IO
[1]   ⍝ FASTER CHARACTER MATRIX SORT.
[2]    ⎕IO←0 ◇ COL←¯1↑⍴MAT ◇ R←⍳1↑⍴MAT
[3]    BASE←2⌈ρCS ◇ MNOC←⌊BASE⍴2*56
[4]   A:→COL↓B ◇ COL←COL-COLS←COL⌊MNOC
[5]    R←R[⍋BASE⊥CS⍳⍉MAT[R;COL+⍳COLS]] ◇ →A
[6]   B:R←MAT[R;]
     ∇
```

Due to internal representation and a phenomenon called "blowup" (wherein four-byte integers must be converted to eight-byte, floating-point format during the base value operation), it is often faster to map only to four-byte integers rather than to eight-byte integers. The only change necessary is to replace the 2*56 on line [3] by 2*31 (or 2147483648). Although more iterations may be necessary, each iteration is less expensive; also, the danger of a *WS FULL* error is reduced. The function *CSORT* (and the related functions *CGRADE*, *CRANK*, *CDRANK*, *CLRANK*, *CHRANK*, and *CARANK*) in workspace 6 *SORT* on the *APL*PLUS* System all use this technique. In addition, they have been optimized to reduce CRU consumption and avoid *WS FULL* situations.

Sorting Individual Rows

Thus far we've discussed matrix sorts where a single logical value (e.g., a name) is represented by several physical values (e.g., a vector of characters in a row). What if we wish to view the matrix as simply a collection of row vectors, each of which we want to sort independently? For numbers, the FORTRAN-style approach would be:

```
     ∇ R←ROWSORT MAT;I;⎕IO
[1]   ⍝ SORTS EACH ROW OF A NUMERIC MATRIX BY ITERATING.
[2]    ⎕IO←1 ◇ →A×⍳I←''⍴⍴R←MAT
[3]   A:R[I;]←MAT[I;⍋MAT[I;]] ◇ →A×⍳I←I-1
     ∇
```

```
      ⎕←M←? 3 6 ⍴9
2 7 5 5 2 1
7 7 9 4 5 8
1 1 5 7 1 4

      ROWSORT M
1 2 2 5 5 7
4 5 7 7 8 9
1 1 1 4 5 7
```

But why loop if we don't have to? Try viewing this problem as a key sort, wherein the major key is the row number of each element and the minor key is the data itself:

```
      □←A←⍴M ◇ □←L←(⍳M),[0.5],⍉(⊖A)⍴⍳1↑A
3 6
 2 7 5 5 2 1 7 7 9 4 5 8 1 1 5 7 1 4      Data.
 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3      Row number.
```

As usual, we sort first by the minor key (the data),

```
      GRM←⍋L[1;] ◇ L[;GRM]
 1 1 1 1 2 2 4 4 5 5 5 5 7 7 7 7 8 9      Data.
 1 3 3 3 1 1 2 3 1 1 2 3 1 2 2 3 2 2      Row number.
```

then by the major key (the row number),

```
      □←L←L[;GRM[⍋L[2;GRM]]]
 1 2 2 5 5 7 4 5 7 7 8 9 1 1 1 4 5 7      Data.
 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3      Row number.
```

then reshape only the first row (the data) back to its original shape,

```
      A⍴L
1 2 2 5 5 7
4 5 7 7 8 9
1 1 1 4 5 7
```

Actually, we need not precompute the row indices because all the information is contained in *A* (the shape) and ⍋,*M* (*GRM*):

```
      GRM,[0.5] ROW←⌈GRM÷1↓A
  6 13 14 17  1  5 10 18  3  4 11 15  2  7  8 16 12  9
  1  3  3  3  1  1  2  3  1  1  2  3  1  2  2  3  2  2
```

Thus all we need to do is rearrange *GRM* based on ⍋*ROW*, and use that result to permute ,*M*.

```
      A⍴(,M)[GRM[⍋ROW]]
1 2 2 5 5 7
4 5 7 7 8 9
1 1 1 4 5 7
```

The completed function would be as follows:

```
     ∇ R←NSORTROW MAT;GRM;□IO
[1]   ⍝ SORTS EACH ROW OF A NUMERIC MATRIX BY PARALLEL PROCESS.
[2]    □IO←1 ◇ R←⍴MAT ◇ MAT←,MAT
[3]    GRM←⍋MAT ◇ R←R⍴MAT[GRM[⍋⌈GRM÷1↓R]]
     ∇
```

The function *NSORTR* in workspace 6 *SORT* uses this approach.

An Unusual Technique

The function *CSORTR* in workspace 6 *SORT* sorts individual rows of a character array. It performs an ascending or descending sort using a global collating sequence *CS*. A description of *CSORTR* can be obtained by executing

```
UTILHOW 'CSORTR'
```

in any Library 6 workspace.

To illustrate how *CSORTR* works, we define *CSORTROW* using the approach taken in *NSORTROW*:

```
     ∇ R←CSORTROW MAT;GRM;□IO
[1]   ⍝ SORTS EACH ROW OF A CHARACTER MATRIX BY PARALLEL PROCESS.
[2]    □IO←1 ◇ R←⍴MAT ◇ MAT←,MAT
[3]    GRM←⍋CS⍳MAT ◇ R←R⍴MAT[GRM[⍋⌈GRM÷1↓R]]
     ∇
```

```
      WEST
HAWAII
ALASKA
WASHINGTON
OREGON
CALIFORNIA
NEVADA

      CS
 ABCDEFGHIJKLMNOPQRSTUVWXYZ

      CSORTROW WEST
    AAHIIW
    AAAKLS
AGHINNOSTW
    EGNOOR
AACFIILNOR
    AADENV
```

However, *CSORTR* actually uses a completely different approach which is distinguished by the fact that it uses very little intermediate storage and is quite fast.

Can you emulate the documented performance of *CSORTR*? To put yourself on the right track, assume that the *M* and *I* keys on your terminal are broken (both uppercase and lowercase), so that you can't use them in your solution. Your solution need only work in origin 1. You can check your algorithm against the solution described in the following article.

March/April 1979

CSORTR Algorithm

The *CSORTR* (from workspace 6 *SORT*) emulation problem appearing at the end of the previous article prompted several expressions of incredulity, particularly regarding the restriction that the *M* and *I* keys could not be used, thus disallowing ⌽, ⍋, and ⍳. The problem was to independently sort each row of a character array based on a collating sequence *CS*. Jak Eskinazi (STSC) and Paul Sobota (STSC) both uncovered the following technique (for origin 1):

```
     ∇ R←A CSORTRX CHR;B;CS;K;K2;L;S
[1]   ⍝ SORTS EACH ROW OF A CHARACTER MATRIX.
[2]    K←' ',(1↑CS[]),1↑0⍴CHR
[3]    K←K[1∊ ¯1 1 ∈A] ◇ L←⍴CS←⍋[,A] CS
[4]    S←⍴CHR ◇ →(1≥L⌊(×/S)⌊¯1↑S)⍴B ◇ K←1
[5]   A:K2←K←K+K ◇ B←CHR∈(L⍴(-K2)↑K⍴1)/CS
[6]    CHR←,CHR,CHR ◇ B←,B,~B
[7]    CHR←S⍴B/CHR ◇ →(L>K←K2)⍴A
[8]   B:R←CHR
     ∇
```

Lines [2] and [3] use implicit error checking (see the article "Error Checking", which appears elsewhere in this book) to ensure that the arguments are valid. Line [4] performs the setup and traps degenerate cases such as an empty right argument. The essential algorithm is contained on lines [5] through [7], which are executed ⌈2⊛⍴CS times (yes, a loop!).

Fundamentally, the technique entails repetitive rearrangement (merging) of the right argument based on the Boolean representation of the positions of the argument's elements in the collating sequence. That is, performing (in origin 1)

```
BR←((⌈2⊛⍴CS)⍴2)⊤¯1+CS⍳CHR
```

gives us a Boolean representation of each element's position in *CS*. If we assume that *CHR* is a vector, and perform

```
CHR←CHR[T←⍋BR[N;]] ◇ BR←BR[;T]
```

for *N* from ⌈2⊛⍴CS to 1 (last row to first), we are essentially performing an ascending radix sort (least significant bit to most significant bit) on the Boolean representation of *CHR*. In *CSORTRX*, *B* on line [5] holds a variant of *BR*[*N*;] at each iteration. The algorithm in *CSORTRX* reverses *CS* and computes the negation of each row of *BR* to reduce the membership (∈) search

time slightly. Because *CSORTRX* must operate on arrays other than vectors, we use a variant of

```
CHR←(⍴CHR)⍴(,(~BR[N;]),BR[N;])/,CHR,CHR
```

on line [7] to rearrange the elements, rather than grade up and subscription.

)ORIGIN 0

This "Whizbang" is different from its predecessors—its examples are written with ⎕*IO* (the index origin) set to 0:

```
      ⎕IO←0 ⋄ ⍳10
0 1 2 3 4 5 6 7 8 9
```

Along the way, we'll discover several practical applications of this useful origin. First of all, only the following primitive functions are affected by the origin:

```
ODR←⍳A                    ODR←A⍳B
ODR←?A                    ODR←A?B
ODR←⍋A                    ODR←⍒A

R←A[OSA]
R←⌽[OSA]A                 R←A⌽[OSA]B
R←A/[OSA]B                R←A\[OSA]B
R←⍉/[OSA]A                R←⍉\[OSA]A
R←A,[OSA]B                R←OSA⍪A
```

The name *ODR* signifies an *origin-dependent result; OSA* signifies an *origin-sensitive argument. ODR* means that the result of the function is affected by the setting of ⎕*IO*; *OSA* means that the range of permissible arguments is affected by the setting of ⎕*IO*.

This distinction between dependent results and sensitive arguments is partially academic, however. Consider the following problem: Generate the value 10 if ⎕*IO* is 1 and ¯10 if ⎕*IO* is 0. Beginners will often struggle and come up with something like this:

```
10××⎕IO-0.5
```
11 characters, 3 functions, ⎕*IO* referenced

An experienced coder might produce:

```
20×⎕IO-0.5
```
10 characters, 2 functions, ⎕*IO* referenced

or better yet

```
¯10×¯1*⎕IO
```
10 characters, 2 functions, ⎕*IO* referenced

Some hotshots may even do some quick analysis and come up with a first-degree polynomial!

```
⎕IO⊥20 ¯10
```
10 characters, 1 function, ⎕*IO* referenced

But an aficionado, one who prizes *APL* for its elegance and terseness, and recognizes the importance of minimizing the costs of solutions programmed in *APL*, would code it as follows:

```
      ⍳0 ¯10[1]          9 characters, 1 function
```

To get back to our original point, although indexing (subscription) is considered to have an origin-sensitive argument but not an origin-dependent result, few would argue that the result of the above case is not origin dependent!

What are some of the less obvious ramifications of origin 0? For one, a trusty standby for branching doesn't work:

```
      →LABEL×⍳CONDITION          For example, →L×⍳M≥C←C+1
```

In fact, we discover that the above expression either branches to 0 or falls through, and the multiplication (and hence the value of *LABEL*) is irrelevant. In origin 1, the value of ⍳1 is 1 (as a one-element vector) and ⍳0 produces an empty vector. But in origin 0, the value of ⍳1 is 0 (again as a vector), while ⍳0 still produces an empty vector. When users catch on to this unpleasantness, they quickly change to

```
      →LABEL⌈⍳CONDITION
```

which works in either origin. We should point out that the expressions →*CONDITION*⍴*LABEL* and →*CONDITION*/*LABEL* also work in either origin, and are usually preferable.

Another strange effect of origin 0 is the behavior of dyadic iota (⍳), particularly with elements not found.

```
      ⍴VEC← 6 2 9 6 10 0
6

      VEC⍳ 0 2 3 6 9
5 1 6 0 2
```

For most of us, the sixth position is *number 6* (that is, we commonly think in origin 1); but in origin 0 the sixth position is *number 5*. Do you remember the rule: "If an element of the right argument to dyadic iota is not found in the left argument (*VEC*), its corresponding position in the result is set as 1+⍴*VEC*."? Well, the correct formula is actually ⎕*IO*+⍴*VEC*. That is, the indices in *VEC* are ⍳⍴*VEC*, and the "not found" flag is actually 1 plus the last index: ¯1↑⍳1+⍴*VEC*.

There are other interesting effects of origin 0. For instance, the results of +/⍳*N* and 2!*N* are identical. To laminate two vectors to form a two-row matrix, we must specify a negative axis!

```
      VEC,[¯0.5]⌽VEC
 6  2  9  6 10  0
 0 10  6  9  2  6
```

Particularly useful is the fact that we can use Boolean results (as from relational functions) to subscript directly, rather than adding 1 (also see the article "Creative Indexing"):

```
      →(L1,L2)[D>⍴VEC]           Branch to either of two lines.

      ' ⎕'[VEC∘.>⍳⌈/VEC]         A histogram.
⎕⎕⎕⎕⎕⎕
⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Behavior with respect to take and drop (↑ and ↓) is modified when their left arguments are the result of origin-dependent functions (dyadic iota in this case):

```
      INP←'PRINT/34'
      ⎕IO←1 ◇ A←(¯1+I←INP⍳'/')↑INP ◇ B←I↓INP
      ⎕IO←0 ◇ A←(I←INP⍳'/')↑INP ◇ B←(I+1)↓INP
      INP ∧.= A,'/',B
1
```

Sometimes it is necessary to write a function that works in either origin, but in which you can't set or localize ⎕IO. For example, a function that returns indices should return them in the origin of the calling environment. Such functions are called *origin sensitive* and are usually annoying to write. Some hints follow:

- If you subscript a vector often, rather than modifying the index at each operation (for instance, using VEC[⎕IO+3] to select the fourth element), consider modifying the vector initially and using fixed origin-1 subscripts:

```
      VEC← 14 13 12 2 8 21 6 10 10 25
      VEC←⎕IO↓0,VEC      or:  VEC←(⍳~⎕IO),VEC
      VEC[4 6 3]
2 21 12
```

- If you refer to counters (C) or limits (L) related to subscripted arrays, remember to adjust the counters and limits for the origin.

  Origin 1:

```
  ⎕IO←1 ◇ L←⍴VEC ◇ R←L⍴¯1 ◇ →A×L≥C←1
A:R[C]←⍴⍴⎕FREAD TN,VEC[C] ◇ →A×L≥C←C+1
```

  Origin 0:

```
  ⎕IO←0 ◇ L←⍴VEC ◇ R←L⍴¯1 ◇ →A×L>C←0
A:R[C]←⍴⍴⎕FREAD TN,VEC[C] ◇ →A×L>C←C+1
```

  Either origin:

```
  L←⍴VEC ◇ R←L⍴¯1 ◇ L←L+⎕IO ◇ →A×L>C←⎕IO
A:R[C]←⍴⍴⎕FREAD TN,VEC[C] ◇ →A×L>C←C+1
```

- If you use the +\BITVEC technique for generating indices, prefix it by BITVEC[⎕IO]←⎕IO to work in either origin.

As a rule, all user-defined functions should have one of the following characteristics:

- origin independent—that is, ⎕IO is localized and set on line [1] to a fixed value;
- origin free—that is, the function does not refer to ⎕IO explicitly or implicitly (uses no origin-sensitive primitives); or
- origin sensitive—as described earlier.

If you must write origin-dependent functions (those that work properly in only one global origin), you should state this limitation prominently in the documentation for the function.

Since the capability to localize the system variable ⎕IO became available, we have observed that many new functions we write have ⎕IO localized and set to 0. Origin 0 often seems more convenient.

For example, whenever dealing with number systems and using primitives like modulus (or residue, |), encode (or representation, ⊤), and decode (or base value, ⊥), origin 0 is the natural base. If a number represents several quantities packed together, its origin-0 representation is consistent with the packed form. In the following example, we want to select several scattered points from a matrix M.

```
      □←M← 4 7 ⍴10×⍳28
  0  10  20  30  40  50  60
 70  80  90 100 110 120 130
140 150 160 170 180 190 200
210 220 230 240 250 260 270

      □←N← 2 1 3 0 ,[¯0.5] 1 3 4 6          ROWS ,[¯0.5] COLUMNS
2 1 3 0
1 3 4 6

      (,M)[□←(⍴M)⊥N]
15 10 25 6
150 100 250 60
```

Since the indices in *N* are already in origin 0, their decoded form is suitable for origin-0 indexing.

Here are some other calculations that are simplified in origin 0:

- Compute the number of days in February (years 1901-2099):

```
29 28 28 28[4|YEAR]        or:  29-×4|YEAR
```

- Branch to different lines based on the rank of an array or the result of □*NC*:

```
→(S,V,M,A)[3⌊⍴⍴ARRAY]
→(A,L,V,F,X)[□NC 'NAME']
```

- Signal only the first line of the diagnostic message if an error occurs:

```
□ELX←'□ERROR(□DM⍳□TCNL)⍴□DM'
```

- Given a matrix *M* with distinct elements, determine, as a two-row matrix, in which row and column certain numbers *N* appear (this is the inverse of the scattered-point example given above):

```
(⍴M)⊤(,M)⍳N
```

  In origin 1 it would be

```
1+(⍴M)⊤((,M)⍳N)-1
```

- Branch to 0 if *'END'* is in *V*; otherwise, fall through:

```
→⍳∨/V □SS 'END'
```

Error Checking

One of the oldest tenets of computer programming and system design is that of GIGO—garbage in, garbage out. If data is accepted with no regard for its validity, erroneous information will surely be entered and subsequent computations and output will reflect and magnify those errors. It is therefore essential that only valid data be accepted for processing and storage.

To ensure that data is valid, the programs that accept the data must verify that it meets a predefined set of criteria. If it passes these tests, it is accepted for subsequent processing; otherwise it is rejected with an indication to the user of what is wrong.

Explicit Error Checking

Most error checking is performed explicitly. The most common mistake in this process is making invalid assumptions. Consider the following code to accept a single integer between 500 and 999, inclusive:

```
[1]  A:I←,⍞ ◇ →(1=⍴I)⍴B
[2]    'ENTER A SINGLE NUMBER.' ◇ →A
[3]  B:→((I≥500)∧(I≤999)∧I=⌊I)⍴C
[4]    'ENTER AN INTEGER FROM 500 TO 999.' ◇ →A
[5]  C:
```

Can you find the invalid assumption here?

The programmer didn't consider the possibility of the user entering character data:

```
⍞:
      '!'
DOMAIN ERROR
FOO[3] B:→((I≥500)∧(I≤999)∧I=⌊I)⍴C
                             ∧
```

Line [1] should have been as follows:

```
[1]  A:I←,⍞ ◇ →((1=⍴I)∧0=1↑0⍴I)⍴B
```

When error checking for the entry of a single integer, I use a program fragment similar to the one that follows.

```
[1]  A:I←,⍞ ◇ →((1≠⍴I)=1↑0⍴I)⍴B
[2]    'ENTER A SINGLE NUMBER.' ◇ →A
[3]  B:→(I<.<(⌈I), 500 1000)⍴C
[4]    'ENTER AN INTEGER FROM 500 TO 999.' ◇ →A
[5]  C:
```

Although slightly unusual, it's recognizable because of the error messages, which act like comments. It uses only 8 functions to perform the tests, rather than the 12 normally used.

Error Checking in Utility Functions

Utility functions are *APL* programs used to perform certain tasks common to many applications, and for which a certain standard of performance is assumed. They are much like *APL* primitive functions, but they usually perform more complicated operations.

One characteristic of a good utility is that, like an *APL* primitive, it be well defined for all possible arguments: acceptable arguments produce the desired results; unacceptable arguments engender appropriate errors. Errors can be signalled to the calling environment so that an application using Exception Handling (⎕*ELX*) can respond appropriately, or they can simply result in suspension of execution.

For instance, the EMMA® utility function *EQ* requires a nine-element numeric vector as its left argument, *F*. The second element must be a tie number, and the ninth element must be a valid field number (a subscript into component 4 (a vector) of the file). Although EMMA utilities do not signal errors, for clarity the following example will use ⎕*ERROR* to signal exceptions. The complete set of tests would be as follows:

```
⎕ERROR(1≠⍴⍴F)/'RANK ERROR'
⎕ERROR(9≠⍴F)/'LENGTH ERROR'
⎕ERROR(0≠1↑0⍴F)/'DOMAIN ERROR'
⎕ERROR(~⎕FAVAIL)/'FILE SUBSYSTEM NOT AVAILABLE'
B←F[2]
⎕ERROR(B≤.≥(⌈B),2* 0 31)/'DOMAIN ERROR'
⎕ERROR(B∧.≠⎕FNUMS)/'FILE TIE ERROR'
⎕ERROR(4≤.≥2⍴⎕FSIZE B)/'FILE INDEX ERROR'
FI←⎕FREAD B,4
⎕ERROR(1≠⍴⍴FI)/'RANK ERROR'
E←F[9]
⎕ERROR(E≤.≥(⌈E), ¯2 2 *31)/'DOMAIN ERROR'
⎕ERROR(E∧.≠⍳⍴FI)/'INDEX ERROR'
E←FI[E]
```

Although relatively expensive to perform, these tests ensure that the argument is acceptable and that it will not produce unpredictable results. Furthermore, the diagnostic messages are recognizable because they emulate those of the *APL***PLUS* System.

Implicit Error Checking

The tests shown above represent explicit error checking. Another technique, actually used in EMMA, is known as implicit error checking. Although often more difficult to formulate, this technique has the advantage of using the built-in error-checking mechanisms of the *APL* system to diagnose and report errors, and is thus considerably faster and terser than explicit error checking. Consider the following statements:

```
B← 0 1 0 0 0 0 0 0 0 1 0 /F[], 4 0
E←(⎕FREAD B)[F[9]]
```

By constructing the *APL* formulation carefully, we need only 6 primitive functions to check implicitly what 34 primitives (plus 10 compressions and calls to ⎕*ERROR*) checked explicitly.

The two statements above will generate one of the following messages:

- *RANK ERROR* if *F* is not a vector (subscription).
- *DOMAIN ERROR* if *F* is not numeric (catenation).

- *LENGTH ERROR* if *F* is not nine elements long (compression).
- *FILE SUBSYSTEM NOT AVAILABLE* if that is the case (⎕*FREAD*).
- *DOMAIN ERROR* if *F*[2] is not a valid tie number (⎕*FREAD*).
- *FILE TIE ERROR* if no file is tied to *F*[2] (⎕*FREAD*).
- *FILE ACCESS ERROR* if access to the file is restricted or requires a passnumber (⎕*FREAD*).
- *FILE INDEX ERROR* if the file does not have a component 4 (⎕*FREAD*).
- *RANK ERROR* if component 4 is not a vector (subscription).
- *DOMAIN ERROR* if *F*[9] is not integral (subscription).
- *INDEX ERROR* if *F*[9] is not a valid index in component 4 (subscription).

This type of checking is used throughout EMMA. Note the additional code for computing *F*[2],4 (*B*); its purpose is to use the *APL* system's internal checking facilities to verify arguments and generate errors in the proper sequence. If a global exception handler were defined as

```
⎕ELX←'⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM'
```

any errors detected would be signalled to the calling environment.

July/August 1979

A Clever Sort

In "String Searching" (see *Collected Whizbangs*, Volume 1), we presented a series of string-searching algorithms culminating in a function (*SS6*), which continually narrowed the search until completion. *SS6* was typically fastest because it searched the fewest possible number of elements at each iteration. Although *SS6* was the most complicated function presented, its algorithmic sophistication outweighed the code complexity.

In this article, we present another such algorithm, this one to grade numeric matrices. The function presented here is derived from a function to grade character matrices written by Dr. Ross Hamsher, then a systems analyst at Thermo Electron Corporation and now with STSC. The fundamental algorithm has been known for many years; Ron Murray wrote a version in *APL* at STSC in 1972. But Ross's function represents a major and elegant simplification of the algorithm, with no loss of generality.

The naive approach to grading a numeric matrix is to repetitively rearrange a permutation vector, starting from the last column (the minor key) and ending with the first column (the major key). For simplicity, we will use origin 1 in *NMGU*.

```
     ∇ R←NMGU MAT;CI;⎕IO
[1]   ⍝ NUMERIC MATRIX GRADE UP.
[2]    ⎕IO←1 ◇ R←⍳1↑⍴MAT ◇ CI←(⍴MAT)[2]
[3]   A:→CI↓0
[4]    R←R[⍋MAT[R;CI]] ◇ CI←CI-1 ◇ →A
     ∇
```

Line [2] computes the initial permutation vector (*R*) and the index of the first column to grade (*CI*). Line [3] escapes if *CI* is 0. Line [4] rearranges the permutation vector according to the next key (column), decrements the column index, and branches back to the test on line [3].

NMGU has the virtue of simplicity, but it is totally insensitive to the characteristics of its argument. For example, consider the following matrix:

```
    MAT
617   267   6864    462   305
301   986   1750   7101  1414
213   340   4611  21243   240
713   691   6319   7000   700
212   751   9305    747     0
312   530   7600   1550     0
301   657   8220   7316   401
```

```
      NMGU MAT
5 3 7 2 6 1 4

      MAT[NMGU MAT;]
  212   751  9305   747     0
  213   340  4611 21243   240
  301   657  8220  7316   401
  301   986  1750  7101  1414
  312   530  7600  1550     0
  617   267  6864   462   305
  713   691  6319  7000   700
```

NMGU dutifully processes all elements of every column. However, without much inspection it's easy to see that we really need only examine the first two columns of *MAT* to determine the grade vector. In fact, the only reason we need to look at the second column is to resolve the duplicate 301 values in the first column.

This technique—sorting each column only to the extent necessary to resolve duplicates in previous columns—is intuitive to most people. It is somewhat more complex to describe.

In the function *NMGRUP* presented below, a Boolean vector (*U*) flags row indices (*R*) of *MAT* whose relative ordering is so far unresolved. A partition vector (*PV*) is used to segment the flagged rows (*MAT*[*U*/*R*;]) into groups of equal value.

At each iteration the value of another column is used to permute unresolved elements of the result within their partition. Typically, this process completely decides some ties and further segregates others. Hence some partitions disappear (become singular) while others break into several smaller partitions. The process continues until either all columns are examined, or all partitions disappear.

The function and a detailed description follow.

```
     ∇ R←NMGRUP MAT;C;D;G;N;U;NC;PV;UR;⎕IO
[1]    ⍝ NUMERIC MATRIX GRADE UP.
[2]    ⎕IO←1 ◇ R←⍳N←1↑⍴MAT ◇ →(NC←(⍴MAT)[2])↓0
[3]    PV←N↑1 ◇ U←N⍴1 ◇ C←1
[4]  A:G←⍋D←MAT[UR←U/R;C] ◇ G←G[⍋(+\PV)[G]]
[5]    R[U/⍳N]←UR[G] ◇ →(NC<C←C+1)⍴0
[6]    D←D[G] ◇ PV←PV∨D≠1⌽D ◇ U←U\UR←PV/⍳1⌽PV
[7]    →A×⍴PV←UR/PV
     ∇
```

Line [2] Set the origin to 1.

◇ Compute the number of rows (*N*) and the tentative result (*R*).

◇ Compute the number of columns (*NC*); quit if there are no columns.

Line [3] Initialize the partition vector (*PV*) to a single encompassing partition.

◇ Initialize the unresolved element flags (*U*) to all 1's.

◇ Initialize the column index (*C*) to 1.

Line [4] *A*: Select unresolved row indices (*UR*) from the result; select the corresponding data elements from the *C*th column (*D*); compute the ascending grade vector (*G*).

◇ Rearrange the grade vector (*G*) according to the partition of each element. *G* now reflects ascending order within each partition.

Line [5] Permute the unresolved elements ($U/\iota N$) of R (UR) to reflect the ordering of the current column.

◇ Increment the column index; quit if all columns have been processed.

Line [6] Rearrange the data elements (D) in ascending order within each partition.

◇ Determine changes in the ordered vector and update the partition vector (PV). Each partition will be split into one or more smaller partitions.

◇ Flag unresolved partitions; that is, those of length two or more (UR). Update the unresolved element flags (U).

Line [7] Select only unresolved partitions; continue processing only if at least one partition remains.

NMGRUP can be further refined by diagnosing some special cases (matrices with less than two rows or columns), and by preprocessing the first column to avoid the generalized partition-processing code in the loop. The refinements affect only the setup (lines [2] through [5] of *NMGRADEUP*); the loop (lines [6] through [9]) is identical to that of *NMGRUP* (lines [4] through [7]).

```
     ∇ R←NMGRADEUP MAT;C;D;G;N;U;NC;PV;UR;⎕IO
[1]   ⍝ NUMERIC MATRIX GRADE UP (WITH SPECIAL CASE HANDLING).
[2]    ⎕IO←1 ◇ N←⍴R←⍋D←(MAT,0)[;1]
[3]    NC←(⍴MAT)[2] ◇ →(2>N⌊NC)⍴0
[4]    D←D[R] ◇ PV←D≠¯1⌽D ◇ U←PV⌿⍳⍴PV
[5]    →(⍴PV←U/PV)↓0 ◇ C←2
[6]   A:G←⍋D←MAT[UR←U/R;C] ◇ G←G[⍋(+\PV)[G]]
[7]    R[U/⍳N]←UR[G] ◇ →(NC<C←C+1)⍴0
[8]    D←D[G] ◇ PV←PV∨D≠¯1⌽D ◇ U←U\UR←PV⌿⍳⍴PV
[9]    →A××⍴PV←UR/PV
     ∇
```

We have added two functions to workspace 6 *SORT* on the *APL*PLUS* System that incorporate Ross's algorithm: *NGRADEUP* and *CGRADEUP* (numeric and character). Experimentation will disclose that with degenerate arguments such as 5 100⍴⍳100, both perform considerably slower than naive algorithms because they have no opportunity to narrow the sort. But with typical corporate data (such as names and addresses) having relatively few duplicated rows and early resolution of partial duplicates, the decrease in computer time can be dramatic. Thank you, Ross, for a clever and inexpensive algorithm!

September/October 1979

Arrays of Arrays

One of the strengths of *APL* is its native ability to manipulate multidimensional arrays. Whereas most other programming languages are basically oriented toward scalars, *APL* is oriented toward rectangular arrays.

Since the early 1970s, language designers (among them Jim Brown, Ziad Ghandour and Jorge Mezei, Trenchard More, and Bob Smith) have been investigating a powerful extension to *APL*, variously called arrays of arrays, general arrays, or nested arrays. Under these proposals, each element of an *APL* array could itself be any array. Thus, a vector would still be a linear collection of elements, but each element could be a vector, a matrix, or whatever. And each element of those subarrays could, in turn, be an array. Boggling!

Actually, these concepts are already familiar to *APL* users. For example, a file may be thought of as a generalized vector wherein each element (component) may be an array of any type, rank, or shape. A matrix of names is really a vector of variable-length vectors, which, to fulfill today's requirement for rectangularity, we pad with trailing blanks.

Using the facilities currently available, let's investigate a common problem in *APL*—that of displaying a list (vector) of character objects (scalars, vectors, or matrices) on a page or screen (a two-dimensional surface, or matrix). Let's assume our page is 24 lines long and 80 characters wide, that we always want to display an entire 1920-character page, and that we're working in origin 1 (`⎕IO←1`).

Scalars

The easiest case is displaying a vector; *APL* handles this problem directly. We have a vector of (nominally `1920 ⍴ 24×80`) scalars—we want a 24 by 80 matrix of scalars. First we must assure enough elements to fill a page: `(×/ 24 80)↑VEC`. Then we simply reshape:

```
24 80 ⍴(×/ 24 80)↑VEC
```

This prints the elements in row-major order; that is, reading from left to right, then down the page. If we wish to display in columns (down the page, then left to right), we would use transpose:

```
⍉ 80 24 ⍴(×/ 24 80)↑VEC
```

Since we'll be dealing with more complex problems later, let's generalize the above two expressions (the underlined portions may be elided):

SCALARS, ACROSS:

```
24 80 ⍴ 1 2 ⍉ 24 80 ⍴(×/ 24 80)↑VEC
```

SCALARS, DOWN:

```
24 80 ⍴ 2 1 ⍉ 80 24 ⍴(×/ 24 80)↑VEC
```

Vectors

The next problem is stated thus: Given a vector of 16-element vectors (for example, names), nominally arranged as a 120 (1920÷16) by 16 matrix, make a 24 by 5 table of names (each a 16-element vector). To display across the page, the solution is easy (see Figure 1 for the result):

```
      ⍴MAT
117 16

      24 80 ⍴((×/ 24 5),16)↑MAT
```

```
ALABAMA          ALASKA           ALBANIA          ALGERIA          ARIZONA
ARKANSAS         AZORES           BAHAMAS          BAHRAIN          BARBADOS
BRUNEI           CALIFORNIA       CAMEROON         CHAD             COLORADO
COMORO ISLANDS   CONGO            CONNECTICUT      CORSICA          CURACAO
DELAWARE         DOMINICA         ECUADOR          EL SALVADOR      ELLICE ISLANDS
ENGLAND          FAROE ISLANDS    FINLAND          FLORIDA          FRANCE
GABON            GAMBIA           GEORGIA          GERMANY          GUATEMALA
GUYANA           HAWAII           HONG KONG        IDAHO            ILLINOIS
INDIANA          IOWA             IRAQ             IRELAND          IVORY COAST
JORDAN           KANSAS           KENTUCKY         KENYA            LIECHTENSTEIN
LOUISIANA        LUXEMBOURG       MADAGASCAR       MAINE            MALAWI
MALI             MALTA            MARYLAND         MASSACHUSETTS    MAURITANIA
MEXICO           MICHIGAN         MINNESOTA        MISSISSIPPI      MISSOURI
MONACO           MONTANA          NEBRASKA         NEVADA           NEVIS
NEW CALEDONIA    NEW HAMPSHIRE    NEW JERSEY       NEW MEXICO       NEW YORK
NIGER            NORTH CAROLINA   NORTH DAKOTA     NORTH KOREA      OHIO
OKLAHOMA         OREGON           PAKISTAN         PENNSYLVANIA     PERU
PORTUGAL         PUERTO RICO      RHODE ISLAND     RHODESIA         SENEGAL
SOMALIA          SOUTH CAROLINA   SOUTH DAKOTA     SPAIN            SUDAN
SURINAM          SYRIA            TENNESSEE        TEXAS            TOGO
TONGA            TRINIDAD         TUNISIA          UPPER VOLTA      UTAH
VERMONT          VIRGINIA         WALES            WASHINGTON       WEST VIRGINIA
WESTERN SAMOA    WISCONSIN        WYOMING          YEMEN            ZAIRE
ZAMBIA           ZANZIBAR
```

Figure 1—Vectors Across a Page

We first assure enough "elements" (each a 16-element vector) to fill the page; then we simply reshape to the desired page size. The row-major ordering implicit in reshape does all we need.

However, to display the vectors down the page, we conceptually need to create a 5 by 24 matrix of 16-element vectors, transpose it, and reshape it to our page size (see Figure 2 for the result):

```
24 80 ⍴ 2 1 3 ⍉ 5 24 16 ⍴((×/ 24 5),16)↑MAT
```

Observe that we started with a matrix, increased its rank by one to express the logical structure (a 5 by 24 matrix of 16-element vectors), interchanged the coordinates representing the "outer" structure (that of a 5 by 24 matrix to that of a 24 by 5 matrix), and finally reduced the rank to that of our medium (a page or screen).

```
ALABAMA          ELLICE ISLANDS   KENYA            NEW JERSEY       SYRIA
ALASKA           ENGLAND          LIECHTENSTEIN    NEW MEXICO       TENNESSEE
ALBANIA          FAROE ISLANDS    LOUISIANA        NEW YORK         TEXAS
ALGERIA          FINLAND          LUXEMBOURG       NIGER            TOGO
ARIZONA          FLORIDA          MADAGASCAR       NORTH CAROLINA   TONGA
ARKANSAS         FRANCE           MAINE            NORTH DAKOTA     TRINIDAD
AZORES           GABON            MALAWI           NORTH KOREA      TUNISIA
BAHAMAS          GAMBIA           MALI             OHIO             UPPER VOLTA
BAHRAIN          GEORGIA          MALTA            OKLAHOMA         UTAH
BARBADOS         GERMANY          MARYLAND         OREGON           VERMONT
BRUNEI           GUATEMALA        MASSACHUSETTS    PAKISTAN         VIRGINIA
CALIFORNIA       GUYANA           MAURITANIA       PENNSYLVANIA     WALES
CAMEROON         HAWAII           MEXICO           PERU             WASHINGTON
CHAD             HONG KONG        MICHIGAN         PORTUGAL         WEST VIRGINIA
COLORADO         IDAHO            MINNESOTA        PUERTO RICO      WESTERN SAMOA
COMORO ISLANDS   ILLINOIS         MISSISSIPPI      RHODE ISLAND     WISCONSIN
CONGO            INDIANA          MISSOURI         RHODESIA         WYOMING
CONNECTICUT      IOWA             MONACO           SENEGAL          YEMEN
CORSICA          IRAQ             MONTANA          SOMALIA          ZAIRE
CURACAO          IRELAND          NEBRASKA         SOUTH CAROLINA   ZAMBIA
DELAWARE         IVORY COAST      NEVADA           SOUTH DAKOTA     ZANZIBAR
DOMINICA         JORDAN           NEVIS            SPAIN
ECUADOR          KANSAS           NEW CALEDONIA    SUDAN
EL SALVADOR      KENTUCKY         NEW HAMPSHIRE    SURINAM
```

Figure 2—Vectors Down a Page

Again, let's generalize the two expressions to see if we can detect any parallels:

VECTORS, ACROSS:

```
24 80 ⍴ 1 2 3 ⍉ 24 5 16 ⍴((×/ 24 5),16)↑MAT
```

VECTORS, DOWN:

```
24 80 ⍴ 2 1 3 ⍉ 5 24 16 ⍴((×/ 24 5),16)↑MAT
```

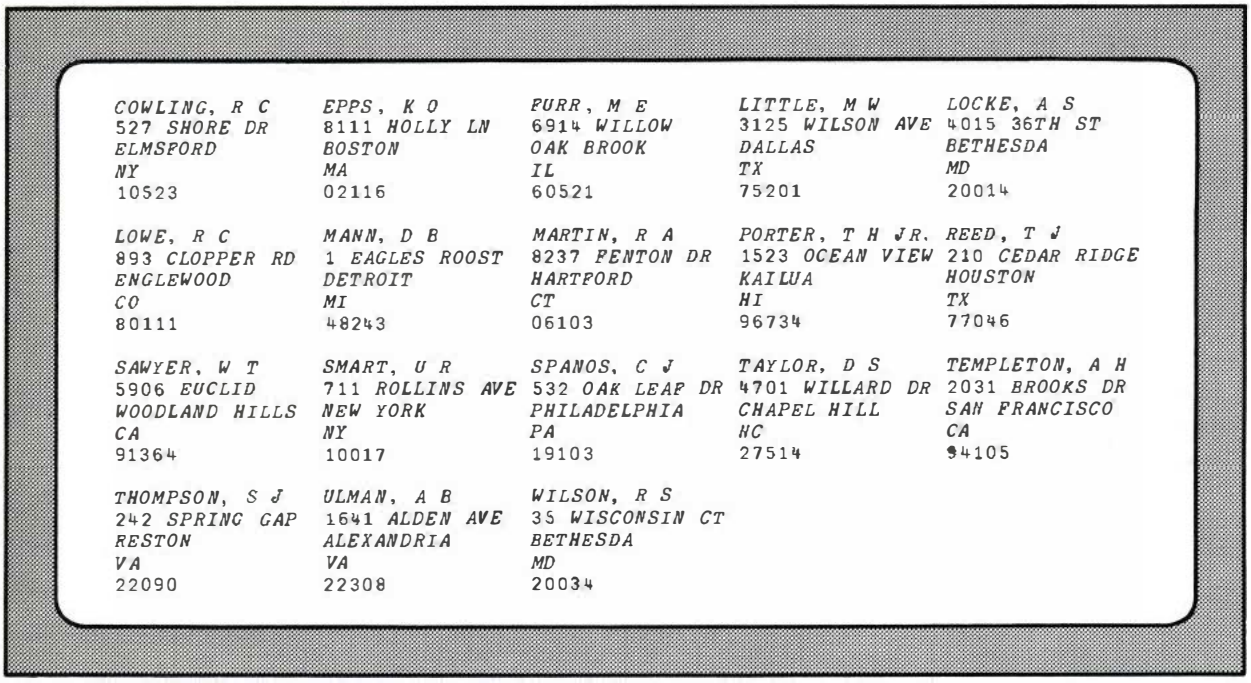
Matrices

The final problem is a natural extension of the first two: Given a 20-element vector of 6 by 16 matrices (for example, addresses), nominally arranged as a 20(1920÷×/ 6 16) by 6 by 16 array, produce a 4 by 5 table of addresses.

This time we are starting with a three-dimensional array whose first coordinate represents the "outer" structure (a 20-element vector) and whose trailing coordinates represent the "inner", or subarray, structure (6 by 16 matrices). Again, to display across the page, we assure enough "elements" to fill the page; then we increase the rank by one to express a new logical structure (a matrix of matrices), interchange the coordinates, and reshape to the page size (see Figure 3 for the result):

```
      ⍴TEN
18 6 16

      24 80 ⍴ 1 3 2 4 ⍉ 4 5 6 16 ⍴((×/ 4 5), 6 16)↑TEN
```

Notice that here the transpose interchanges the coordinates of the layers and rows because we want layers adjacent rather than rows; otherwise, the rows of each address would be spread across the page.

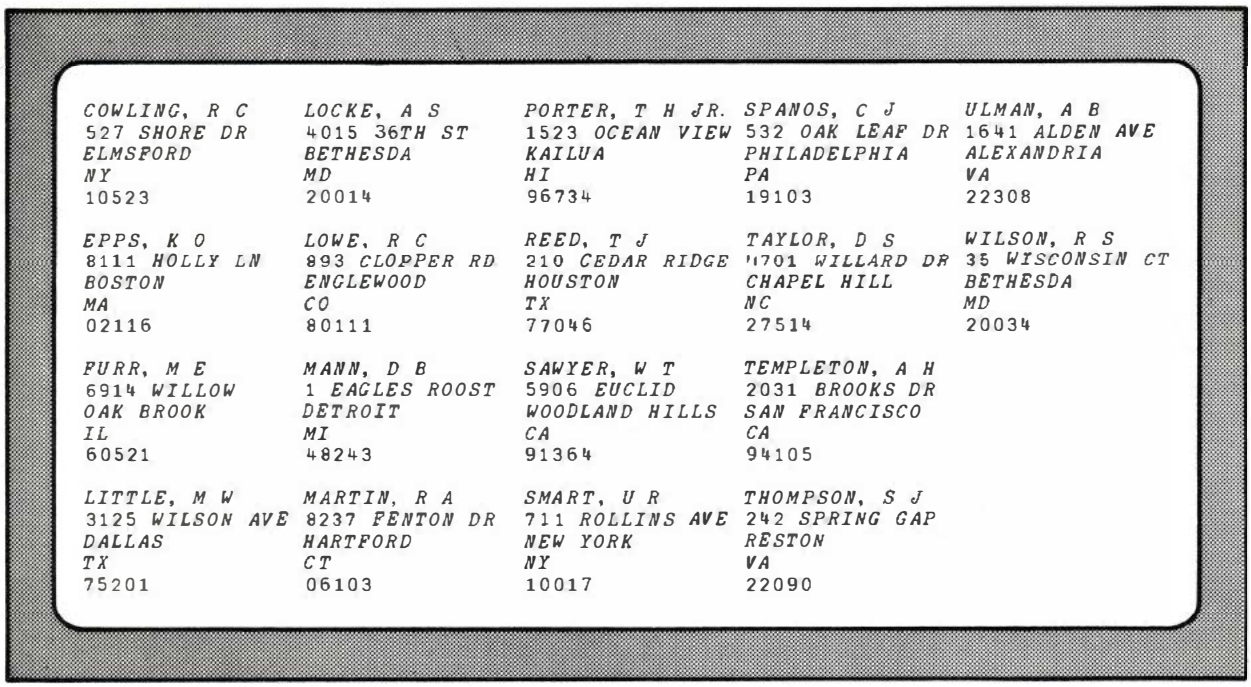


```
COWLING, R C      EPPS, K O       FURR, M E       LITTLE, M W     LOCKE, A S
527 SHORE DR      8111 HOLLY LN   6914 WILLOW     3125 WILSON AVE 4015 36TH ST
ELMSFORD          BOSTON          OAK BROOK       DALLAS          BETHESDA
NY                MA              IL              TX              MD
10523             02116           60521           75201           20014

LOWE, R C         MANN, D B       MARTIN, R A     PORTER, T H JR. REED, T J
893 CLOPPER RD    1 EAGLES ROOST  8237 FENTON DR  1523 OCEAN VIEW 210 CEDAR RIDGE
ENGLEWOOD         DETROIT         HARTFORD        KAILUA          HOUSTON
CO                MI              CT              HI              TX
80111             48243           06103           96734           77046

SAWYER, W T       SMART, U R      SPANOS, C J     TAYLOR, D S     TEMPLETON, A H
5906 EUCLID       711 ROLLINS AVE 532 OAK LEAF DR 4701 WILLARD DR 2031 BROOKS DR
WOODLAND HILLS    NEW YORK        PHILADELPHIA    CHAPEL HILL     SAN FRANCISCO
CA                NY              PA              HC              CA
91364             10017           19103           27514           94105

THOMPSON, S J     ULMAN, A B      WILSON, R S
242 SPRING GAP    1641 ALDEN AVE  35 WISCONSIN CT
RESTON            ALEXANDRIA      BETHESDA
VA                VA              MD
22090             22308           20034
```

Figure 3—Matrices Across a Page

To display the matrices down the page, we perform the same operations, again interchanging the leading two coordinates of the reshape and the transpose (see Figure 4 for the result):

```
24 80 ⍴ 3 1 2 4 ⍉ 5 4 6 16 ⍴((×/ 4 5), 6 16)↑TEN
```



```
COWLING, R C      LOCKE, A S      PORTER, T H JR. SPANOS, C J     ULMAN, A B
527 SHORE DR      4015 36TH ST    1523 OCEAN VIEW 532 OAK LEAF DR 1641 ALDEN AVE
ELMSFORD          BETHESDA        KAILUA          PHILADELPHIA    ALEXANDRIA
NY                MD              HI              PA              VA
10523             20014           96734           19103           22308

EPPS, K O         LOWE, R C       REED, T J       TAYLOR, D S     WILSON, R S
8111 HOLLY LN     893 CLOPPER RD  210 CEDAR RIDGE 4701 WILLARD DR 35 WISCONSIN CT
BOSTON            ENGLEWOOD       HOUSTON         CHAPEL HILL     BETHESDA
MA                CO              TX              NC              MD
02116             80111           77046           27514           20034

FURR, M E         MANN, D B       SAWYER, W T     TEMPLETON, A H
6914 WILLOW       1 EAGLES ROOST  5906 EUCLID      2031 BROOKS DR
OAK BROOK         DETROIT         WOODLAND HILLS  SAN FRANCISCO
IL                MI              CA              CA
60521             48243           91364           94105

LITTLE, M W       MARTIN, R A     SMART, U R      THOMPSON, S J
3125 WILSON AVE   8237 FENTON DR  711 ROLLINS AVE 242 SPRING GAP
DALLAS            HARTFORD        NEW YORK        RESTON
TX                CT              NY              VA
75201             06103           10017           22090
```

Figure 4—Matrices Down a Page

Let's summarize the expressions to display matrices:

MATRICES, ACROSS:

```
24 80 ⍴ 1 3 2 4 ⍉ 4 5 6 16 ⍴((×/ 4 5), 6 16)↑TEN
```

MATRICES, DOWN:

```
24 80 ⍴ 3 1 2 4 ⍉ 5 4 6 16 ⍴((×/ 4 5), 6 16)↑TEN
```

Generalizing

Of course, a truly general-purpose function would not have the constants (such as 24 80 and 6 16) embedded in the code, and might accept vectors of multidimensional arrays. Further, provision would have to be made for margins, subarrays which don't fit neatly onto (i.e., evenly divide) a page, and varying page sizes. You might wish to try your hand at such a function. Workspace 444 *LABELS*—derived from 444 *ADDRFP*, which is described in Supplement No. 2 to the *APL*PLUS Address Management Package User's Guide* (STSC, 1977)—illustrates how one such system handles the problem.

November/December 1979

Expansive Thinking

When IBM released the first implementation of *APL* in 1968, they gave it the name *APL\360*. Computer people understood that 360 referred to the then-current IBM 360 series of computers, and learned that *APL* stood for A Programming Language. But that funny slash (\)—now wasn't that backwards?

Of course, *APL* devotees chuckled and understood—it meant *APL expanded* the power of IBM 360 computers. (We suppose that STSC chose its trademark, *APL*PLUS*, for similar reasons—STSC's enhancements raised the power of *APL*.) Thus expansion was the first function particular to *APL* to which many people were exposed. Unfortunately, newcomers to *APL* don't have the benefit of such symbolism in the names of most current *APL* systems. It is for them that we present this article on expansion.

Expansion, like its companion function compression, accepts any array (*ARR*) as its right argument, a Boolean vector (*BV*) as its left argument, and optional use of the axis operator ([*K*]), for which we'll assume origin 1 (⎕*IO*←1):

R←*BV**ARR*	Last coordinate, same as *BV*\[⍴⍴*ARR*]*ARR*.
R←*BV*⌿*ARR*	First coordinate, same as *BV*\[1]*ARR*.
R←*BV*\[*K*]*ARR*	*K*th coordinate, general form.

The result is the same as *ARR* except that the *K*th (or last or first) coordinate is expanded to length (⍴*BV*). Positions corresponding to 1's in *BV* are filled with data from *ARR*; positions corresponding to 0's in *BV* are filled with 0's (for numeric data) or blanks (for character data):

```
      1 0 0 0 1 0 0 1 0 1 0 \ 11 22 33 44
11 0 0 0 22 0 0 33 0 44 0

      W←'ANTEPENULTIMATE'
      1 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 \W
AN TE PEN  UL  TI MATE

      1 0 1 1 0 \⍳5
LENGTH ERROR
      1 0 1 1 0 \⍳5
                ^
```

Since the 1's dictate where the data from the right argument is positioned in the result, there must be exactly as many 1's in the left argument as there are elements in the right argument. More formally, for *R*←*BV*\[*K*]*ARR*:

```
(+/BV)=(⍴ARR)[K]
(⍴R)∧.=((K-1)↑⍴ARR),(⍴BV),K↓⍴ARR
```

If the right argument is a scalar, it is extended to a vector of the proper length (+/*BV*):

```
      APL← 0 0 1 1 0
      APL\360
0 0 360 360 0
```

Expansion is a natural partner to compression. Compression "undoes" what expansion does, but not vice versa. That is, (except for scalars) *A* is identical to *B*/*B**A*. But the expression *B**B*/*A* fills the result with 0's (or blanks) in positions corresponding to 0's in *B*, thus making *B**B*/*A* identical to *B*×*A* for numeric vectors. For matrices we can set rows (or columns) to 0 based on a condition.

```
      MAT
 1  1  2  1  1  1  2  3  4  3
 3  2  1  1  3  5  1  3  3  5
 4  3  5  4  5  1  2  4  1  1
 1  1  3  1  2  1  2  1  2  5
 4  1  2  2  2  1  1  2  2  1
 5  3  1  5  3  1  3  2  3  3

      +/MAT                          Row totals.
19  27  30  19  18  29

      □←COND←20<+/MAT                Row totals exceed 20.
0 1 1 0 0 1

      COND⌿COND≠MAT                  Retain only rows whose total exceeds 20;
 0  0  0  0  0  0  0  0  0  0        make all others 0.
 3  2  1  1  3  5  1  3  3  5
 4  3  5  4  5  1  2  4  1  1
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 5  3  1  5  3  1  3  2  3  3
```

The above expression is clearer than the corresponding multiplication (shown below)

```
MAT×⍉(⍴⍴MAT)⍴COND
```

and has the advantage of also working with character arrays (blanking rows corresponding to 0's in *COND*).

More commonly we wish to perform an operation only on selected elements of an array, and substitute a "fill" value (which might not be 0 or blank) for other elements. For example, suppose that we are given several employee numbers as *EN* and that we want to print the corresponding names and social security numbers.

```
      EN
1158  1626  1272  99999  1778  1363
```

We have available a master vector of all employee numbers *ENOS*, a corresponding matrix of employee names *ENAS*, and a vector of social security numbers *SSEC*.

```
      'I4,X2,6A1,X2,I9' □FMT (ENOS;ENAS;SSEC)
1158  PERRY    980304871
1230  BRENT    876235613
1272  HOLTON   231879651
1363  WILKES   128623497
1514  GOLDEN   123123123
1778  REGIS    878787878
1863  PIERCE   222334444
1881  JOHNS    122567290
1955  MIX      547940666
1978  SMITH    123456789
```

The fundamental algorithm

```
      I←ENOS⍳EN
      ⎕←RPT←ENAS[I;],'G< 999-99-9999>' ⎕FMT SSEC[I]
INDEX ERROR
      ⎕←RPT←ENAS[I;],'G< 999-99-9999>' ⎕FMT SSEC[I]
                                       ^
```

fails if any of the employee numbers in *EN* is not found in *ENOS* (1626 and 99999 in this case).

Since we really wish to select and format only valid employee numbers, we adjust the algorithm to use only those indices that exist (*E/I*), using expansion to leave a blank line for invalid employee numbers.

```
      I←ENOS⍳EN
      E←I≤⍴ENOS ◇ I←E/I
      ⎕←RPT←E\ENAS[I;],'G< 999-99-9999>' ⎕FMT SSEC[I]
PERRY  980-30-4871

HOLTON 231-87-9651

REGIS  878-78-7878
WILKES 128-62-3497
```

If we wish, we can further refine the report to include a diagnostic message for invalid employee numbers by using the negation of *E* to identify the rows of *RPT* that were left blank by the expansion.

```
      N←(~E)/⍳⍴E
      RPT[N;]←'⍰ NOT FOUND: ⍰,LI6' ⎕FMT EN[N]
      RPT
PERRY  980-30-4871
 NOT FOUND: 1626
HOLTON 231-87-9651
 NOT FOUND: 99999
REGIS  878-78-7878
WILKES 128-62-3497
```

The general form of such exception processing for vectors is as follows (assume *PROCESS* and *EXCEPT* process data to produce the result):

```
      RESULT←E\PROCESS E/DATA
```

and optionally

```
      N←(~E)/⍳⍴E
      RESULT[N]←EXCEPT DATA[N]
```

Expansion is normally viewed as using the left argument to modify the right argument. Sometimes the reverse is true, with the right argument being used to replace 1's in the left argument. Consider the following problem: Given the ⎕*VR* form of a function *VRSELECT*, you wish to select certain lines *LN*.

```
    ∇ R←LN VRSELECT VR;PV;⎕IO
[1]   ⍝ SELECTS UNDUPLICATED LINES IN ORDER FROM ⎕VR FORM OF FUNCTION.
[2]    ⎕IO←1 ◇ →(⍴R←¯6↓VR)↓0
[3]    PV←¯1⌽R ⎕SS ⎕TCNL,'[' ◇ PV[1]←1
[4]    LN←(⍳+/PV)∈LN+1
[5]    PV[PV/⍳⍴PV]←LN≠¯1↓0,LN
[6]    R←(≠\PV)/R
    ∇

      0 4 5 VRSELECT ⎕VR 'VRSELECT'
    ∇ R←LN VRSELECT VR;PV;⎕IO
[4]    LN←(⍳+/PV)∈LN+1
[5]    PV[PV/⍳⍴PV]←LN≠¯1↓0,LN
```

A partition vector *PV* identifies the beginning of each line (a bracket preceded by a new-line character). The line numbers are transformed into a Boolean

vector indicating, for each line of the function, whether it is to be selected (1) or not (0). The tricky code is on line [5]; we modify *PV* so that the first and last+1 element of each contiguous set of selected lines remains marked with a 1, but all other lines become unmarked. Then we use not-equal scan (≠\) to mark odd parity, which selects the proper lines. Using expansion, line [5] becomes simply

```
[5]    PV←PV\LN≠¯1↓0,LN
```

thus using its right argument to modify the left!

The last technique we will discuss is a traditional use of expansion to form a rectangular *APL* array from a vector of delimited substrings. Variously known by names such as *VTOM*, *SSMAT*, *MIM*, *ROWNAMES*, and *BOX*, this algorithm uses a raveled Boolean histogram to supply the proper number of fill elements to expand the vector so that each substring is extended to the same length. By convention the delimiter is the first element of the vector.

```
     ∇ M←VTOM V;DEL;LEN;HIST;INDS;⎕IO
[1]    ⎕IO←1 ⍝ SIMILAR TO <SSMAT> (WS '6 INPUT').
[2]    INDS←(DEL←V=1↑V)/⍳LEN←⍴V
[3]    LEN←(1↓(INDS-1),LEN)-INDS
[4]    HIST←LEN∘.≥⍳0⌈⌈/LEN
[5]    M←(⍴HIST)⍴(,HIST)\(~DEL)/V
     ∇
```

Line [2] determines the indices of the delimiters. Line [3] computes the lengths of the substrings. Line [4] generates a minimal-width histogram, using 1's to mean "select another element from *V*", and 0's to mean "fill with blanks (or 0's)". The expansion and reshape on line [5] use that (raveled) histogram to expand the non-delimiting elements of *V* into rows of the resulting matrix.

```
      M←VTOM '/BRENT/GOLDEN/HOLTON/JOHNS/MIX'
      ⍴M
5 6

      M
BRENT
GOLDEN
HOLTON
JOHNS
MIX
```

January/February 1980

Rotation

The symbol for rotate in *APL*, ⌽, evokes the image of a child's toy top, or of a globe with the axis of rotation drawn in. It is formed by overstriking the circular ○ and modulus | symbols. Rotation treats a vector as a continuous chain of values, which it shifts (cycles) according to some modulus of the left argument.

Essentially, rotate shifts a vector of values (specified in the right argument) a number of places (specified in the left argument):

```
      V←'RESPECT'
      0⌽V ◇ 2⌽V ◇ ¯1⌽V
RESPECT
SPECTRE
TRESPEC
```

As you can see, if the left argument is positive, the elements are shifted to the left (beginning to end) and if the left argument is negative, the elements are shifted to the right (end to beginning). Of course, if the left argument is 0, no shifting is performed at all. But what of large left arguments? For example,

```
      125⌽'EIGHTH'
HEIGHT
```

Does *APL* actually tediously shift the vector 125 positions?

Consider the question: "What is the position of the connecting link in a bicycle chain having 60 links after rotating the chain 1253 links?" Rather than rotating the chain, counting off 1253 links one by one, we realize that the first 1200 link movements are superfluous, and that all we need do is rotate the chain 53 links and note the final position of the connecting link. In *APL* nomenclature, we would rotate `60|1253` links.

Rotate (*N*⌽*V*) likewise considers its right argument a circular chain. If the right argument has ⍴*V* elements, the chain has ⍴*V* "links"; in the question above we considered 60 links. Thus, rotating the chain *N* links is identical to rotating it `(⍴V)|N` links (`6|125`, or `5` in the *'EIGHTH'* example):

```
N⌽V  ←→  ((⍴V)|N)⌽V
```

This expression maps all left arguments (including negative numbers) into a canonical form between 0 and `(⍴V)-1`. Alternatively, if you prefer rotation to the right, you could map the left arguments into a canonical form between `1-⍴V` and 0 by

```
N⌽V  ←→  ((-⍴V)|N)⌽V
```

Thus in the bicycle chain example above, the clever solution (least link movements counted) would be to rotate the chain *backward* 7 links rather than forward 53 links!

```
      60 ¯60|1253
53 ¯7
```

We should note that *APL* uses the optimal data movement strategy internally, so you need not worry about reducing the left argument to any minimal form.

In practice, the (by far) most common left arguments to rotate are 1 and ¯1, used in differencing. Some examples follow (all in origin 1):

- *CUM* represents the running balances in your checkbook. Compute the transaction amounts (deposits are positive, withdrawals and checks are negative):

```
TRANS←CUM-¯1φCUM ◇ TRANS[1]←CUM[1]
```

  which can be read as: "Subtract from each element of *CUM* the previous element."

```
      2▼CUM,[0.5] TRANS
 400.00  300.00  250.00  200.05  700.05
 400.00 ¯100.00  ¯50.00  ¯49.95  500.00
```

  Alternatively,

```
TRANS←CUM-¯1↓0,CUM
```

- *CV* is a character vector. Delete all leading, trailing, and multiple adjacent blanks:

```
CV←' ',CV ◇ NB←CV≠' '
CLEAN←1↓(NB∨1φNB)/CV
```

  which can be read as: "Select from *CV* elements that are nonblank or that are followed by nonblanks."

```
      '|',CV,'|' ◇ '|',CLEAN,'|'
|  DELETE   ALL  EXTRA BLANKS   |
|DELETE ALL EXTRA BLANKS|
```

  Alternatively,

```
CV←' ',CV ◇ B←CV=' '
CLEAN←1↓(B∧1φB)/CV
```

  which can be read as: "Select from *CV* elements which are not both blank and followed by a blank."

- Mark the beginning of each series of identical elements in the (nonempty) vector *V*:

```
B←V≠¯1φV ◇ B[1]←1
```

  which can be read as: "Where do elements in *V* differ from the previous element?"

```
      V ◇ B
2 2 3 4 4 5 5 5 6 6 7 7 7 8
1 0 1 1 0 1 0 0 1 0 1 0 0 1
```

- Mark the end of each series of identical elements:

```
B←V≠1φV ◇ B[ρB]←1
```

  which can be read as: "Where do elements in *V* differ from the following element?"

```
      V ◇ B
2 2 3 4 4 5 5 5 6 6 7 7 7 8
0 1 1 0 1 0 0 1 0 1 0 0 1 1
```

As you may have gathered, 1⌽ can be read as "the following (or next)" and ¯1⌽ can be read as "the previous (or last)".

Another common application of rotate is to align matrices. This use derives from the ability of rotate to shift many vectors (collected in a matrix or higher-dimensional array) at once:

```
      MAT
MANGER
SELVES
PUTOUT

      3⌽MAT       or:  3 3 3⌽MAT
GERMAN
VESSEL
OUTPUT

      MAT0
SPRITE
RECENT
LAYOUT
ENLIST

      ¯1 2 3 ¯4⌽MAT0
ESPRIT
CENTRE
OUTLAY
LISTEN
```

For example, left justification means rotating an array to the left until a "significant" element (typically a nonblank) is encountered:

```
      '|',CM,'|'
|      THIS     |
|   MATRIX       |
|         WILL BE|
|LEFT JUSTIFIED  |

      +/∧\CM=' '      Count the number of leading blanks.
7 3 9 0

      LJ←(+/∧\CM=' ')⌽CM

      '|',LJ,'|'
|THIS            |
|MATRIX          |
|WILL BE         |
|LEFT JUSTIFIED  |
```

As an exercise, try the following problem in alignment and justification. The answer is given in the next article.

You are given the ⎕CR (matrix) form of an *APL* function that is (of course) less than 100 lines long. Your job is to convert it to the ⎕VR (vector) form, visually identical to the output produced by ∇*FUNCTION*[⎕]∇. You may assume that the function contains no new-line characters (carriage returns) embedded in character constants. The essential problem is to exdent labels and comments one column (or indent all other lines one column).

Here's a skeleton of the function:

```
    ∇ R←CRTOVR FN;⎕IO;CR;ALP;LOC;LINES
[1]   ⍝ <FN> IS THE NAME OF AN EXISTING FUNCTION;
[2]   ⍝ <R> IS THE ⎕VR FORM OF THAT FUNCTION.
[3]    ⎕IO←1 ◇ CR←⎕CR FN ◇ →(0∈⍴CR)↓A
[4]    'FN LOCKED, NONEXISTENT, OR SHADOWED' ◇ →0
[5]   A:ALP←CR∈'ABCDEFGHIJKLMNOPQRSTUVWXYZΔABCDEFGHIJKLMNOPQRSTUVWXYZΔ0123456789'
[6]   ⍝ COMPUTE WHICH LINES ARE LABELS OR COMMENTS:
[7]    LOC← ⍝ YOUR CODE GOES HERE!
[8]   ⍝ USE <LOC> TO INDENT <CR> APPROPRIATELY:
```

```
[9]   CR← ⍝  YOUR CODE GOES HERE!
[10] ⍝ FORMAT LINE NUMBERS:
[11]  LINES←'P<[>Q<]>LI5' ⎕FMT⍳¯1+1↑⍴CR
[12] ⍝ ADD LINE NUMBERS AND ∇ DECORATIONS TO <CR>:
[13]  CR←('     ∇',[1] LINES,[1] '     ∇'),CR,[1] ' '
[14] ⍝ ADD NEW-LINE CHARACTERS, RAVEL, AND REMOVE
[15] ⍝ ALL TRAILING BLANKS:
[16]  R←(,(⍲∨\⌽CR≠' '),1)/,CR,⎕TCNL
    ∇
```

The function is located in workspace 901 *WHIZBANG* on the *APL⋆PLUS* System. Fill in the code required on lines [7] and [9].

March/June 1980

Rotation Problem—Results

In the previous article on rotation, you were left with a problem—to supply the code to compute and exdent lines beginning with a label or comment in the □*CR* (matrix) form of an *APL* function. The code you supplied should resemble that on lines [7] and [9] of the following function (available in workspace 901 *WHIZBANG* on the *APL*PLUS* System):

```
    ∇ R←CRTOVR FN;□IO;CR;ALP;LOC;LINES
[1]  ⍝ <FN> IS THE NAME OF AN EXISTING FUNCTION;
[2]  ⍝ <R> IS THE □VR FORM OF THAT FUNCTION.
[3]   □IO←1 ◇ CR←□CR FN ◇ →(0∈⍴CR)⍴A
[4]   'FN LOCKED, NONEXISTENT, OR SHADOWED' ◇ →0
[5]  A:ALP←CR∈'ABCDEFGHIJKLMNOPQRSTUVWXYZΔABCDEFGHIJKLMNOPQRSTUVWXYZΔ0123456789'
[6]  ⍝ COMPUTE WHICH LINES ARE LABELS OR COMMENTS:
[7]   LOC←(CR[;1]∈'⍝')∨((+/∧\ALP)⌽CR)[;1]=':'
[8]  ⍝ USE <LOC> TO INDENT <CR> APPROPRIATELY:
[9]   CR←LOC⌽' ',CR
[10] ⍝ FORMAT LINE NUMBERS:
[11]  LINES←'P<[>Q<]>LI5' □FMT⍳¯1+1⍴⍴CR
[12] ⍝ ADD LINE NUMBERS AND ∇ DECORATIONS TO <CR>:
[13]  CR←('    ∇',[1] LINES,[1] '    ∇'),CR,[1] ' '
[14] ⍝ ADD NEW-LINE CHARACTERS, RAVEL, AND REMOVE
[15] ⍝ ALL TRAILING BLANKS:
[16]  R←(,(⌽∨\⌽CR≠' '),1)/,CR,□TCNL
    ∇
```

Creative Indexing

Most programmers think of indexing as a rather mundane function, used only to select elements, rows, and columns of arrays. As "Subscription" (see *Collected Whizbangs*, Volume 1) pointed out, subscription is actually a powerful tool to identify elements of an array for selection or replacement. This article presents some unusual applications of indexing.

Origin 0 (See also the article "`)ORIGIN 0`".)

When `⎕IO` is set to 0, indexing can be used directly with subscripts of 0—Boolean values, for example. The statement

```
→(L1,L2)[M>C]
```

branches to *L2* if *M* is greater than *C*, and to *L1* otherwise. Likewise, the statement

```
→(L0,L1,LN)[2⌊⍴V]
```

branches to *L0* if *V* is an empty vector, to *L1* if *V* has one element, and to *LN* otherwise.

Few Elements, Many Indices

Notice how the following function *HIST* subscripts a four-element vector with a much larger matrix of indices. Origin 0 is again used to avoid the annoying subtraction of 1 from *YEAR* and the addition of 1 to the indices.

```
    ∇ R←YEAR HIST REVEXP;M;⎕IO
[1]   ⍝ PRODUCES A PROFIT (LOSS) HISTOGRAM.
[2]   ⍝ <YEAR> IS THE BEGINNING YEAR.
[3]   ⍝ <REVEXP> IS A 2-ROW MATRIX:  FIRST IS REVENUE;
[4]   ⍝ SECOND IS EXPENSE.  THE VALUES MUST BE ONE- OR
[5]   ⍝ TWO-DIGIT POSITIVE INTEGERS.
[6]    ⎕IO←0
[7]    R←'I4,X1' ⎕FMT YEAR+⍳1↓⍴REVEXP
[8]    R←R,' *O●'[2⊥REVEXP∘.>⍳M←⌈/,REVEXP]
[9]    M←¯4+⍳M+5
[10]   R←R,[0]⍉'BLI2' ⎕FMT M×0=5|M
    ∇

      REV ◇ EXP
4  5  7  10  14  17  21
4  6  6   8  12  14  18
```

```
      1974 HIST REV,[0.5]EXP
1974 ⍟⍟⍟⍟
1975 ⍟⍟⍟⍟⍟*
1976 ⍟⍟⍟⍟⍟⍟o
1977 ⍟⍟⍟⍟⍟⍟⍟⍟oo
1978 ⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟oo
1979 ⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟ooo
1980 ⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟⍟ooo
          5     1     1     2
                0     5     0
```

Slicing Transpose to Refine the Result

As the original article on subscription suggests, one way to select scattered elements from an array is to refine the result of indexing by using dyadic transpose (⍉) to select a diagonal.

In the function that follows we use subscription first to select the digit indices (line [5]), then to select the proper order-of-magnitude character set (line [6]). By this time we have too much data, so we select the major diagonal of each plane and transpose it (line [7]). Note that `1 0 0⍉R` is identical to `⍉0 1 1⍉R`. Finally we ravel and remove excess blanks.

```
     ∇ R←ROMAN A;⎕IO
[1]   ⍝ FORMAT AS A ROMAN NUMERAL ONE
[2]   ⍝ INTEGER IN THE RANGE 0-3999.
[3]    ⎕IO←0
[4]    R← 0 16 20 22 24 32 36 38 39 28
[5]    R←(0 4 2 2⊤R)[;4 10 10 10⊤A]
[6]    R←(¯4↑4↑3 4⍴'MCXI DLV ')[R;]
[7]    R←,1 0 0⍉R◇R←(R≠' ')/R
     ∇

      ROMAN 1444
MCDXLIV

      ROMAN 1666
MDCLXVI

      ROMAN 2507
MMDVII

      ROMAN 3888
MMMDCCCLXXXVIII
```

This function is particularly instructive, not only because of its powerful use of subscription and transpose, but also because of its uses of encode (⊤) on line [5]. A good exercise to understand the result of `0 4 2 2⊤R` is to hand compute the operation. Observe that origin 0 is again more appropriate, this time because we're indexing by the digits of various numbering systems, all of which start with 0.

Precomputed Translation

There are several special circumstances where we can use subscription on a precomputed array to reduce processing time significantly. For example, if *V* is a long vector of integers in the range 1 to 15, it may be faster to perform `(⍟⍳15)[V]` than to calculate `⍟V`. Rather than computing the natural logarithm of every element of *V*, we simply compute the 15 possible result values and select the appropriate elements. Subscription is used to *translate* from one range of values to another. The technique can be modified to work with noninteger values by adding a "lookup" vector of the unique input values:

```
PRECOMPUTEDΔOUTPUTS [POSSIBLEΔINPUTS ⍳ INPUT]
```

Removing Duplicate Values

Suppose we have several thousand account numbers *ACCTS* (integers in the range 1001-9999) from which we wish to remove duplicates. The traditional and most general technique,

```
UNIQ←((ACCTS⍳ACCTS)=⍳⍴ACCTS)/ACCTS
```

is currently far too expensive (approximately $6.00). A refined technique,

```
SORTED←ACCTS[⍋ACCTS]
UNIQ←(¯1↓1,SORTED≠1⌽SORTED)/SORTED
```

is much less expensive (approximately $0.40), but still takes no advantage of an important characteristic of *ACCTS*, namely that it is integer with a known and relatively small range of values.

The "Whizbang technique" is to create an incidence vector, mark the elements corresponding to the given account numbers, and then note the positions of the marked elements:

```
MAX←9999
IV←MAX⍴0 ◇ IV[ACCTS]←1 ◇ UNIQ←IV/⍳MAX
```

Observe that a given element of *IV* may be marked (set to 1) several times; of course re-marking an element that is already marked has no further effect. The cost: about ten cents!

Fast Epsilon

In certain cases of membership, we can profitably use an incidence vector as a precomputed translation if we know that the data has a limited range. For example, assume that *A* and *B* are both large vectors of account numbers like those referred to in the preceding section. *A* represents all accounts, and *B* represents accounts opened before 1979. We wish to flag in *A*, those accounts opened before 1979. While $A \in B$ is terse, the following statements can be considerably faster.

```
MAX←9999
IV←MAX⍴0 ◇ IV[B]←1 ◇ FLAG←IV[A]
```

We again create an incidence vector and mark the elements corresponding to accounts opened before 1979. We then use that incidence vector as a translation vector for the accounts in *A*.

A related technique is described in *DETAIL* for the variable *BITDATES* in workspace 6 *DATES* on the *APL*PLUS* System. *BITDATES* is a 123199-element Boolean incidence vector that has been pre-marked with all valid dates from January 1, 1900, to December 31, 1999. Thus, subscripting *BITDATES* by dates in the form *MMDDYY* provides an extremely fast verification technique (possibly in conjunction with STSC's Exception Handling Facility). You could further modify *BITDATES* to flag weekdays, holidays, anniversaries, fiscal periods, or whatever you can precompute.

Inverting a Permutation

If *PV* is a permutation vector, then *IP*←⍋*PV* computes its inverse permutation, as does *IP*←*PV*⍳⍳⍴*PV*. Considerably faster than both of these algorithms is the following:

```
IP←⍳⍴PV ◇ IP[PV]←IP
```

Thus, to rank a numeric vector *V* (using the permutation ⍋*V*), the expression

```
R←⍳⍴V ◇ R[⍋V]←R
```

is almost twice as fast as $R←⍋⍋V$, and is usually many times faster than $R←(⍋V)⍳⍳⍴V$.

Often it is convenient to process data in ascending order, but the result is needed in the same order as the original data. The initial processing is something like:

```
DATA←DATA[GV←⍋DATA]
RESULT←PROCESS DATA
```

Then the result is rearranged back to the original order using the inverse to the initial permutation:

```
RESULT←RESULT[⍋GV]
```

Instead, we can take advantage of subscripted assignment to invert the permutation of elements back to their original order, thus saving a grade operation:

```
RESULT[GV]←RESULT
```

For example,

```
     ∇ R←DENSERANK V;P
[1]    ⍝ PRODUCES A DENSE RANK OF NUMERIC VECTOR <V>
[2]    ⍝ (LIKE:  ⎕IO←1 ◇ 1 DRANK V
[3]    ⍝ IN WORKSPACE '6 SORT' ON THE APL*PLUS SYSTEM).
[4]     V←V[P←⍋V] ⍝ ARRANGE <V> IN ASCENDING ORDER.
[5]    ⍝ MARK FIRST OCCURRENCE OF EACH DISTINCT ELEMENT.
[6]     V←¯1↓1,V≠1⌽V
[7]    ⍝ COMPUTE IN WHICH PARTITION EACH ELEMENT OCCURS.
[8]     R←+\V
[9]    ⍝ REARRANGE IN ORIGINAL ORDER (SAME AS R←R[⍋P]).
[10]    R[P]←R
     ∇

      GOLFSCORES,[0.5] DENSERANK GOLFSCORES
 76 66 68 76 71 77 67 73 67 71 67 70 67 73 71
  7  1  3  7  5  8  2  6  2  5  2  4  2  6  5
```

Miscellaneous Techniques with Elided Subscripts

How can we assure an argument is a vector, forcing failure if it is not (see the section "Implicit Error Checking" in the article "Error Checking")? Try $A←A[\,]$, which uses very little time or storage (no copy of A is made). Use $A←(1/A)[\,]$ to also allow (and ravel) scalars. To assure a matrix, use $A['';'']$. (Unfortunately, the current implementation of $A[;]$ busily indexes each and every element of A.)

Suppose A is a numeric array and you want to assure it is stored as compactly as possible by *APL* without losing any precision.

```
B←(⍴V←,A)⍴0 ◇ B[]←V ◇ A←(⍴A)⍴B
```

The above statements demote the internal form to the minimum required to represent the data in A.

ρ*A*+*B* Solution

The ρ *A*+*B* problem, presented at the beginning of this volume, prompted a wide variety of imaginative responses ranging from 10 to 84 characters. Many did not work, apparently because the conformability requirements for scalar dyadic functions* were not clearly understood. *A*+*B* will work if

```
(1∧.=ρA) ∨
(1∧.=ρB) ∨
((ρρA)=ρρB)∧(ρA)∧.=(ρρA)↑ρB
```

That is, no length or rank errors will occur if either argument to a scalar function is a singleton (a one-element array of any rank). Did your solution work for 2 3 4 + 1 1 ρ 7? If not, you were probably assuming that the shape of the result would be the shape of the argument having the highest rank—not always true! The rule is

```
ρA+B   IS   ρB
       IF   1≠×/ρB
       OR   1=×/ρB  AND  1=×/ρA  AND  (ρρB)>ρρA
     ELSE   ρA
```

Martin Gardner (STSC) and Paul Penfield (MIT) said it best:

```
((~1∈ρ,B)/ρB),(1∈ρ,B)/(ρA),(1∈ρ,A)/(ρρA)↑ρB
```

* Incidentally, there are 21 scalar dyadic functions. We tend to think of them in the following memory-jogging groups:

Arithmetics (4)	+ × − ÷
"Round" ones (3)	○ ⋆ ⊛
"Tall" ones (4)	| ⌊ ⌈ !
Relationals (6)	< ≤ = ≥ > ≠
Logicals (4)	∧ ∨ ⍲ ⍱

NOTES

NOTES

Collected Whizbangs,
Volume 2

We write our publications to help you. You can help us by filling out and returning FEEDBACK. Your reply will be forwarded to the person who can best benefit from your comments.

1. Is the writing straightforward and clear? Yes ☐ No ☐
2. Is the material organized well? Yes ☐ No ☐
3. Are the examples clear and well chosen? Yes ☐ No ☐
4. Do the illustrations (figures and tables) clarify the material? Yes ☐ No ☐
5. Do you find the size of the publication and the way it is bound convenient? Yes ☐ No ☐
6. Are there errors, or misleading information, in the publication? If so, where do they occur? Yes ☐ No ☐
7. Did you find any parts of the publication unduly hard to understand? If so, which are they? Yes ☐ No ☐

How can we change this publication to make it more useful to you? (If you need more room, please write on the back.)

Thank you for your comments. No postage is necessary if mailed in the U.S.A.

CUT HERE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #10586 Rockville, MD

Postage Will Be Paid By Addressee

STSC, Inc.
2115 E. Jefferson Street
Rockville, Maryland 20852

CUT HERE

FOLD

FOLD, TAPE, AND MAIL (DO NOT STAPLE)

stsc

STSC, Inc.
2115 E. Jefferson Street
Rockville, Maryland 20852
301-984-5000

P049-0781